

OBJECT-ORIENTED DESIGN FOR MOLECULAR MODELING:
FRAMEWORK FOR APPLICATION DEVELOPMENT

BY

JAE-ICK OH

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1996

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor and chairman, Dr. John Staudhammer, for his guidance and encouragement throughout this research. In particular, I am greatly indebted to my cochairman, Dr. Paul Chun, Department of Biochemistry and Molecular Biology, for the invaluable advice and financial support to start working on this dissertation. His stringent instruction and progressive guidance in biochemical issues made this work possible.

I also want to thank all the members of this committee, Professor Stanley Su and Professor Antonio Arroyo of the Department of Electrical and Computer Engineering, Professor Panos Livadas of the Department of Computer and Information Science and Engineering, for their recommendation and advice regarding this dissertation.

I express my sincere appreciation to my wife for her dedicated support through the years of the preparation of this dissertation. Finally, I would like to express my appreciation for the funding provided by the Division of Sponsored Research, University of Florida.

TABLE OF CONTENTS

ACKNOWLEDGEMENT	ii
ABSTRACT.....	v
CHAPTERS	
1 INTRODUCTION	1
2 BACKGROUND	6
2.1 Software Development Techniques	6
2.1.1 Software Reuse	6
2.1.2 Object-Oriented Technology.....	8
2.1.3 Object-Oriented Framework	10
2.1.4 Design Pattern.....	12
2.1.5 Component Technology.....	13
2.2 Domain Analysis.....	14
2.2.1 Protein Structure	15
2.2.2 Protein Modeling	16
2.2.3 Survey of Molecular Modeling Systems.....	17
3 OVERVIEW OF OBJECT-ORIENTED TOOLBOX FOR MOLECULAR MODELING	21
3.1 Introduction.....	21
3.2 Molecular Modeling Foundation Class: MMFC	23
3.3 Three-Dimensional Protein Analysis Program: Pro3D.....	25
3.3.1 Visual Programming	25
3.3.2 Component Software	26
3.3.3 Structure Viewer	26
4 MOLECULAR MODELING FOUNDATION CLASS.....	27
4.1 Introduction.....	27
4.2 Data Model.....	28
4.2.1 Molecular Components.....	29
4.2.2 Objects Relationships and Collaborations	34

4.2.3. Structure Generation and Modification.....	36
4.3 Iterator.....	39
4.4 Algorithm.....	42
4.4.1 Generator.....	45
4.4.2 Predicate.....	49
4.4.3 Mathematical Functions.....	51
5 APPLICATION DESIGN AND VISUAL PROGRAMMING.....	54
5.1 Introduction.....	54
5.2 Visual Programming.....	55
5.3 Module, Port and Connector.....	57
5.3.1 Module.....	57
5.3.2 Port.....	59
5.3.3 Connector.....	60
5.3.4 Execution of Module Network.....	61
6 STRUCTURE MANIPULATION AND VISUALIZATION.....	64
6.1 Introduction.....	64
6.2 Molecular Graphics.....	66
6.2.1 Visualization of Molecular Structure.....	66
6.2.2 Visualization of Molecular Properties.....	68
6.3 Molecular Query Method.....	68
6.4 Applications.....	71
7 MOLECULAR SIMULATIONS.....	77
7.1 Introduction.....	77
7.2 Force Fields.....	78
7.2.1 Potential Energy.....	79
7.2.2 Gradients.....	82
7.2.3 Force-Field Function Objects.....	84
7.3 Energy Minimization.....	86
7.3.1 Steepest Descent Method.....	87
7.3.2 Conjugate Gradient Method.....	88
7.3.3 Adopted-Basis Set Newton-Raphson Method.....	90
7.3.4 Minimizer.....	90
8 CONCLUSION.....	95
REFERENCES.....	98
BIOGRAPHICAL SKETCH.....	103

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

OBJECT-ORIENTED TOOLBOX FOR MOLECULAR MODELING:
FRAMEWORK AND APPLICATION DESIGN

By

Jae-Ick Oh

December 1996

Chairman: Dr. John Staudhammer

Cochairman: Dr. Paul W. Chun

Major Department: Electrical and Computer Engineering

In recent years, a number of molecular modeling systems have been developed. Most systems are monolithic programs developed using conventional programming techniques, which has resulted in software systems that are complicated and hard to maintain and extend. Today, technologies and environments for software development are changing rapidly. Object-oriented and component technology are two primary technical vehicles in modern software development. One of the main advantages of object-oriented technology is that it supports a high degree of software reuse through a framework. A framework predefines the overall structure of the system and captures the components that are common to its application domain so that the application designer can concentrate on the specifics of his application, minimizing the necessary effort. The Internet and the World Wide Web are becoming an important development environment for new applications, revolutionizing the way scientific research is performed. The combination of such new technologies and environments revolutionizes the way applications are developed,

distributed, and used. It is highly desirable to design an infrastructure for scientific applications to best exploit these new opportunities.

In this research, we have developed an Object-oriented Toolbox for Molecular Modeling (OTMM). The core of OTMM is a domain-specific, object-oriented framework that is designed to support a broad range of applications for molecular modeling. While the primary benefits of frameworks are directed at programmers by providing reusable classes which can be used at compile-time, our framework goes one step further by expanding it to exploit the component software technology, and brings the benefits directly to end-users as well as developers. Component software allows reusable objects to be bound at run time and addresses the critical needs to smoothly integrate components written by different developers and upgrade software without disrupting the operation of a current system. Another important feature of OTMM is a visual programming style user interface that allows a user to visually build sophisticated application procedures. We demonstrate the usage of OTMM by presenting important applications in the area of molecular modeling: a molecular graphics system, a molecular simulation system, and a user interface system.

CHAPTER 1

INTRODUCTION

Investigations into the properties of biological molecules such as proteins and DNA include the analysis of the conformational structures, energies, properties, and interactions. In the past, most attempts to explain properties and interactions of biological molecules have been based on experimental work. Today, with the rapid advances in computer technology, molecular modeling [5, 18] on a computer has proven a powerful tool for the study of molecular structure, making it possible to simulate and analyze the structure of small and medium-sized molecules through theoretical simulations [13, 17, 44] and molecular graphics [7, 35, 59]. The primary purpose of a molecular modeling program is to build a model of the structure of interest, perform various simulations on that structure, and provide powerful graphics and computational tools which allow a researcher to efficiently visualize and analyze the structural and dynamic properties of the molecule.

In recent years, a number of molecular modeling systems have been developed. Software systems such as MIDAS [27, 28], RIBBONS [12], RASMOL [64], and MOLSCRIPT [47] provide various structural representations of molecules focusing on the molecular graphics. Software systems such as AMBER [60], CHARMM [8], and GROMOS [73] incorporate computational chemistry methods including energy minimization and molecular dynamics to study the dynamic properties of molecules.

Most earlier systems are monolithic programs designed using conventional programming techniques. Traditional methods based on procedural or structural design

treat dynamic behaviors of molecules independent of the structural representation, and are not suitable for the design of a complex and dynamic molecular modeling system. In addition, application development has generally focused on algorithmic advances of the underlying application, providing rich sets of features but no easy way to add missing features or remove unneeded ones. Such programs contain few reusable components so that existing functionalities cannot easily be reused in other software developments requiring similar functionality.

Today, technologies and environments for software development are changing rapidly. Object-oriented technology and component technology are two primary techniques in modern software development. An object-oriented environment is well suited to model and manipulate complex macromolecular structures. Objects hide their internal algorithms and data, and communicate with each other through well-defined interfaces to ensure the encapsulation of the data manipulation.

Furthermore, object-oriented programming supports a high degree of software reuse through the mechanism of a framework. A framework predefines the overall structure of the system and captures the components that are common to its application domain so that the application designer can concentrate on the specifics of his application. The framework provides application developers with an ideal environment, one which is flexible and extendible without requiring unnecessary effort. A domain-specific framework addresses a design problem within a particular domain such as the graphical user interface [50, 74], 3-D graphics system [69, 78, 79], data processing [26], multimedia [31, 66], or communication software [38]. Component software allows reusable objects to be bound at run-time and addresses the critical needs to smoothly integrate components written by different developers and upgrade software without disrupting the operation of a currently distributed system.

The Internet and intranets are becoming an important development environment for new applications. The Internet and the World Wide Web (WWW) are revolutionizing the way scientific research is performed. Applications and data aren't confined to stand-alone processors any more. The Internet allows scientists to more easily share data and collaborate with each other, making it possible to exchange and reference information from anywhere on the Internet.

The combination of such new technologies and environments revolutionizes the way applications are developed, distributed, and used. It is highly desirable to design an infrastructure for scientific applications to best exploit these new opportunities. In this research, we have developed an Object-oriented Toolbox for Molecular Modeling (OTMM) for creating a new generation of high quality and high performance scientific applications. The OTMM unifies the state-of-the-art software technologies into one framework as a development environment for creating flexible, modular, and integrated applications. It combines various new technologies--object-oriented framework, design pattern, visual programming, and support for a component technology. The core of the OTMM are the Molecular Modeling Foundation Classes (MMFC) that make up a domain-specific, object-oriented framework designed to support a broad range of applications for molecular modeling including computational chemistry, molecular simulation, structural visualization and manipulation. It supplies software development toolkits for users to employ in creating their own applications rather than providing complete software packages, thus encouraging users to modify, extend, and add functions as required.

While the primary benefits of such frameworks are directed at programmers by providing reusable classes which can be used at compile-time, the OTMM goes one step further by expanding it to exploit the component software technology and brings the

benefits directly to end-users as well as developers. On the developer's side, new functions can be easily implemented in the form of components with the MMFC. The components will all inter-operate with the same look and same feel because they are built in the OTMM environment. This advancement will give developers the ability to partition complex scientific applications at will, and permits collaboration between scientists in different locations with different areas of expertise. The Internet allows the automatic distribution of such components.

Visual programming is an intuitive process where users graphically connect application components to build a visualization network. This feature benefits users by providing a novel user interface to visually build sophisticated application procedures. This user interface environment is well suited to molecular modeling applications, where functions run in succession - the output from one becomes the input for the next. The visual programming approach coupled with object-oriented framework and component software allows users to assemble programs with exactly the functionality they need.

This thesis is organized as follows. Chapter 2 reviews the background of modern software technologies on which this research is based. It also provides a survey of related work. Since the key to developing the successful framework lies in understanding and defining the application domain, we collect as much information as possible on the application domain by reviewing existing systems. Chapter 3 provides an overview of the OTMM. OTMM is composed of two primary components: MMFC and Pro3D. MMFC is the framework and Pro3D is an application program built on the MMFC where custom-built application components can be integrated and run. This chapter describes the architecture of MMFC and Pro3D briefly, introducing their various technologies. In chapter 4, we describe the structure and design of the MMFC, detailing all the features and concepts involved in the MMFC.

Developing applications using the framework is as important as developing a framework itself. While the framework is primarily concerned with the programmer's interface to the tools, this visual programming style user interface facilitates the complex process of molecular modeling and brings benefits of the framework to end-users as well. Chapter 5 discusses the features of OTMM that support component-based application development and visual programming. In chapter 6, the visualization of various structures and properties of molecules is described. This application demonstrates the power of the framework with the structural manipulation of complex macromolecules. In chapter 7, the development of a molecular mechanics and dynamics system is described. An important characteristic of scientific software systems designed for a domain like molecular modeling is that they contain complex mathematical algorithms which are constantly updated and newly developed. This application demonstrates how the framework helps to model and implement such algorithms. Finally, chapter 8 summarizes the major accomplishments and suggests directions for future research. With the popularity of the Internet, the network-centric program environment is becoming increasingly important. In order to take full advantage of the Internet, component software in this research can be expanded to more advanced features such as Web-enabled components and distributed components.

CHAPTER 2 BACKGROUND

The Object-oriented Toolbox for Molecular Modeling (OTMM) is a domain-specific, integrated software development environment, which is a result of interdisciplinary efforts that require a fair understanding of concepts in structural and molecular biology as well as a software engineering. In this chapter, we will review various software technologies on which this research is based, present a basic biochemical background of macromolecules, and describe existing molecular modeling systems.

2.1 Software Development Techniques

As application software systems have grown significantly in both complexity and size, it has become important to maintain and upgrade such systems efficiently. Today, technologies and environments for software development are rapidly changing. In this section, we review some important aspects and technologies in modern software development.

2.1.1 Software Reuse

Software reuse [11, 30, 80] is the process of creating software systems from existing components, rather than building from scratch. Software reuse allows one to develop similar applications for many different users without developing the same solutions over and over again. Reusing code that already exists, rather than writing every application from scratch, speeds development and reduces the cost of writing and maintaining an application.

Even if the advantages of reuse are obvious, reusable software components are not common because developing software for reuse is much more difficult and takes longer than developing single-purpose software. There are many technical obstacles to the process. It is almost impossible for a reusable code designer to know everything about the context in which his reusable code will be used, since user requirements often conflict. Designers of reusable code must be able to predict future changes and produce a design that can evolve to meet those changes without causing problems for users.

The main task involved in developing software for reuse is to identify potential reusers with similar requirements, analyze the variations among their requirements, and design a general solution which can be easily adapted to satisfy as many requirements as possible. As different users have different requirements for the same component, it is better to construct components that can be easily adapted by different users. This means that components should be as general as possible. We must, however, also consider the cost of adapting and integrating a general component.

Two conflicting techniques for representing reusability associated with this issue are *widening* and *narrowing*. Widening means identifying a set of requirements, then making a general component that satisfies all of them. The advantage with this approach is that components can be reused *as is* in most cases. The disadvantage is that it is expensive to develop all the functionalities at once, and the code may contain many functions unnecessary to some users which waste space and degrade efficiency. In narrowing, we identify functionalities common to several users which can be represented by an abstract component. Users build actual requirements by extending and specializing the abstract component. This issue is one of the major design trade-offs to be considered by the reusable software developer.

Today, there exist two primary techniques to support software reuse: object technology and component technology. Object technologies are targeted at compile-time reuse, while component technologies are targeted at run-time reuse. These two approaches are intended for different levels of reuse and may be used to complement each other.

2.1.2 Object-Oriented Technology

Object-oriented design and programming [6, 63] is useful for building self-contained, custom applications by creating object definitions in the form of object classes. These language-based object definitions can be shared and reused in different applications, binding them to applications at compile-time. The major concepts involved in object-oriented design are encapsulation, data abstraction, inheritance, and polymorphism. Objects combine a hidden state and operations to be performed on that state, which is represented as a set of object attributes. Every operation declared by an object specifies the objects it takes as parameters and the operation's return value. This is known as the operation's signature. The set of all signatures defined by an object's operations is called the *interface* to the object. Objects are known only through their interfaces, hiding all its implementation details. This feature of objects is called *encapsulation*. The fact that objects are encapsulated produces another important feature which is called *data abstraction*. Abstraction is the process by which complex ideas and structures are made more understandable by the hiding of detail and the generalization of their behavior.

Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship. When a class B has been defined as a subclass of a class A, B inherits all of A's behavior, data and methods. The point of creating a subclass is to add some additional behavior besides that which is inherited. Functions specific to the subclass can be written by simply adding the part that is specific to the subclass, the rest

is inherited automatically. Class inheritance provides a simple and powerful reuse mechanism for defining new classes that inherit properties from existing classes.

Another important aspect of class inheritance is to define families of objects with identical interfaces. An abstract class is one whose main purpose is to define a common interface for its subclasses. All classes derived from an abstract class will share its interface. Some or all of its implementation to operations will be defined in subclasses. All subclasses can then respond to the requests in the interface of this abstract class. A class in which all the member functions are purely virtual and contains no member data is called a pure abstract class. The sole purpose of this class is to define interfaces that must be overridden by the derived class. In general abstract classes are designed to reflect the common behaviors of their derived classes.

Dynamic binding is the ability to link at runtime to the code providing a method's implementation. In C++ [3, 11, 21, 53, 54, 67, 70], this mechanism is implemented through *virtual function*. A virtual member function indicates that if the function is called for an instance of a derived class that is only known as being an instance of the base class, the member function of the derived class is called. This results in *polymorphic* member functions, that can be called without specifying the object's exact type. In C++, polymorphism means that a method's signature may be the same for a number of different classes which belong to the same inheritance structure, while each class contains different implementations for the method.

The relationships between objects are described by generalization/specialization, aggregation and association. A component is specialized for adaptation by the class inheritance. Specialization is also defined as an *is-a* relationship. Aggregation is the relationship in which an object has another object as a component part, and is defined as a *has-a* relationship. The association relationship is present if a member function uses

another class and it is neither an aggregation nor a generalization/specialization relationship. This is called a *uses-a* relationship.

By encapsulating data and functions in a single entity, object-oriented programming makes it possible to break down monolithic applications into functional modules, or components, that can be added or removed as they are needed. Moreover, object-oriented programming can potentially reduce the expense of implementing new objects through mechanisms such as inheritance, polymorphism, and aggregation, to make use of the capabilities built into already-existing objects.

2.1.3 Object-Oriented Framework

One of the greatest advantages of object-oriented programming is that it achieves a higher degree of software reuse through the concept of a framework [80]. The concept of the framework originated in the domain of GUIs (Graphic User Interface).

Model/View/Controller [46] and MacApp [2] are among the first widely used frameworks in this area. Several frameworks have been developed to address a design problem within a particular domain such as the graphical user interface [50, 74], 3D graphics system [69, 78, 79], data processing [26], multimedia [31, 66], operating system, or communication software [38]. However, reusable domain-specific frameworks are still relatively rare, particularly in the realm of scientific software systems such as molecular modeling.

An object-oriented framework is a type of class library with a collection of tightly coupled interconnecting classes and the interfaces that provide an infrastructure for the application developer. The purpose of the framework is to reduce the effort required to design applications by taking advantage of the many built-in and reusable components in the framework. The developer writes only the code that extends the framework behavior to suit the application's requirement, and can focus on the particular problem domain

avoiding the problems and overhead associated with designing all of the program structure and flow of control.

A key benefit of a framework is that it not only provides a large body of prebuilt functionality but offers an architecture in which to add new functionality. The flow of control between the framework and the application is bi-directional. Through the use of dynamic binding, the framework can make calls back to the application. This feature enables the framework to include general functionality where only the extensions have to be specified by the application. An elegant architecture gives developers a logical and obvious location to add application-specific code when implementing features in their applications. A framework provides a uniform, well-defined program structure and data model that enhances compatibility between different applications. Applications independently developed using the same framework can be seamlessly integrated into a single system.

A framework is customized to a particular application by creating application-specific subclasses. A key problem in designing a framework is how to factor out the structure and behavior common to various applications. The framework should represent an application domain abstraction in a generalized and adaptable fashion. The most challenging design trades-offs for framework developers lie in determining which components in a framework should be variable and which should be stable. Insufficient variation makes it hard for users to customize framework components. This results in a framework that cannot accommodate the functional requirements of diverse applications. Conversely, insufficient stability makes it hard for users to comprehend and depend on the framework's behavior. This results in a framework that is not efficient and not able to satisfy the performance requirements.

Interface is the most important aspect of framework design. Interface design and functional factoring constitute the key intellectual content of a framework. Abstract classes are an important part of the framework design because they not only define behavior that is shared by many classes, they also provide interfaces for their subclasses. An implementation of an abstract class includes two different types of methods termed base methods and abstract methods. *Base methods* provide default implementation that is generally useful to subclasses and can be inherited by subclasses. *Abstract methods* have no implementation body and must be overridden by subclasses. The sole purpose of this method is to specify the interface for its subclasses. All methods intended to be redefined in subclasses must be declared as virtual. Defining methods as virtual tells a reuser which methods are intended to be redefined and allows the use of polymorphism.

2.1.4 Design Pattern

The reuse of design is more important than the reuse of code. Although classes in object-oriented programming provide a way to express the program design, they are too fine-grained to efficiently represent the overall design structure. A complex system requires many levels of abstraction, one nested within the other. A design has groups of classes that collaborate to fulfill a larger purpose.

Design patterns proposed by Gamma [30] are sets of such collaborating classes to fulfill a common set of responsibilities. Design patterns capture existing, well-proven design experience established by experts over time. They provide the static and dynamic structures of solutions that occur repeatedly in developing applications. Patterns can be used as building blocks with abstractions above the level of classes to compose the architecture of a software system. They are useful not only in designing a framework but also in describing that design.

Applying design patterns requires a fair knowledge of both design patterns and their relationships to the application. Many domain-independent, tactical patterns are well documented in [22, 30] and are reused rather than reinvented. New patterns have been developed that are specific to the application domain.

2.1.5 Component Technology

Object-oriented programming languages do not provide a means for separate applications to be integrated with other applications. These languages require that when an object is upgraded, dependent applications should be recompiled and redistributed simultaneously. This is unrealistic in a distributed system, where components are typically supplied by different sources.

Component software offers a more efficient and productive model for the software design, reducing the programming effort and time required to deliver new functions. Users can manipulate objects no matter who designed the component applications or what language or development tool was used to program the software components. Through a standard programmatic interface based on the system object standard, components can communicate with each other and be integrated into a complete line of solutions. In addition, components are prefabricated and compiled, which means that users never need to understand the component's implementation details in order to use the component.

With a component, you can bind to these objects at run-time as long as the component observes the standards of component usage. The component itself can be physically staged either on the same machine, or a separate server on the network. In all cases, the user of the component is completely unaware of this physical packaging and staging. The primary benefits of object-oriented programming are directed at programmers, while component software brings the benefits of object technology directly to end-users as well as developers. Component software addresses the following critical needs:

- The need to smoothly integrate components written by different developers using different environments.
- The need for an overall object model to facilitate component communication across application and machine boundaries, that is, across a network.
- The need to enhance and upgrade software components autonomously, without disrupting the operation of a currently distributed system

2.2 Domain Analysis

Developing software development kits requires a mature understanding of the application domain. Domain analysis is an attempt to identify the classes and objects and relationships that are common to all applications within a given domain. The processes involved include defining the domain, collecting applications in the domain, analyzing sample applications and developing models and components. The analyst should collect knowledge from as many different sources as possible and produce a consistent and sufficiently complete model of the domain to understand and extract key abstractions and common mechanisms that serve all the different kinds of applications in the domain. The focus should be upon modeling the domain in its entirety, not just an application within the domain.

Determining the scope of the domain is an important issue in domain analysis. Molecular modeling involves a huge area of applications and it should be clearly defined how general a framework will be before designing that framework. The framework in this work is specifically designed for developing a wide area of applications for biological macromolecules such as proteins and DNA. The philosophy is to identify abstractions that solve extensive classes of problems in this area.

2.2.1 Protein Structure

The basic building unit of a protein is a peptide residue made up of 20 amino acids. An α -amino acid consists of an amino group, a carboxyl group, a hydrogen atom and a distinctive side chain bonded to an α -carbon atom [65]. Various amino acids result from the variation of their side chains in shape, size and polarity. Shape and size affect the packing together of amino acids in the final molecule. Polarity determines the nature and strength of interactions between amino acids in a protein and between the protein and water. In aqueous solution, polar residues tend to be hydrophilic while nonpolar residues tend to be hydrophobic. Thus, polar residues prefer water and tend to surface while nonpolar residues tend to bury themselves inside the molecule.

The backbone of a protein is a linked sequence of amino acids which form a polypeptide chain. An amino acid unit in a polypeptide is called a residue. Proteins commonly consist of from 50 to 500 amino acids which correspond to 500 to 5,000 atoms. The sequence of amino acids, which is the primary structure of proteins, determines the structure and properties of the protein. The polypeptide chain is intrinsically flexible because many of the covalent bonds are rotationally permissive. We can specify a polypeptide conformation by the torsion angles about the $C_\alpha - N$ bond (ϕ) and the $C_\alpha - C$ bond (ψ) as shown in Figure 2.1. The primary limitations to the ranges of rotation for ϕ and ψ arise from nonbonded interactions which include van der Waals forces, electrostatic and hydrogen bonding.

The polypeptide chain of a given type of protein folds compactly into a native three-dimensional structure in aqueous solution [61]. The native structure represents an efficient packing of atoms determined by their attractions for each other in the solvent surroundings. Most of the backbone of the compact, native molecule can be divided into regions of secondary structure, which are distinct segments having characteristic shapes.

The secondary structures fall into three main categories: alpha helices, beta strands or beta sheets, and turns connecting the helices and strands [48]. In beta strands the backbone is extended out while in beta sheets two or more parallel or antiparallel strands are arranged in rows. Secondary elements can combine with one another to form motifs, or supersecondary structures, and the final assembly of all secondary elements is the tertiary structure.

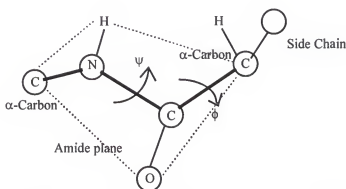


Figure 2.1. The torsional degree of freedom of a peptide unit

Determining the three-dimensional conformational structure of the protein has been one of the most important subjects in the field of protein study. Understanding the 3D structure of proteins is critical to studying their functions and to the potential design of new proteins for biological and medical purposes.

2.2.2 Protein Modeling

In modeling a protein, three principal steps are involved:

- 1) Construct the 3D structure of a protein.
- 2) Predict the 3D structure of the new protein by force field calculations, including energy minimization and molecular dynamics.
- 3) Analyze and visualize the structure and properties of the new protein.

Protein modeling in general can be divided into two functional approaches: energy based and knowledge based. These approaches are usually complementary and are often

used in combination. Energy-based modeling involves an attempt to predict the overall three-dimensional structures of globular proteins by seeking a structure with the minimum energy using mathematical optimization algorithms. Though this method is relatively simple and easy to use, it cannot be used to find the global energy minimum.

Only a local minimum will be found, that is the one closest to the starting point.

Knowledge-based modeling attempts to use what is known about protein structure in general to predict or fit specific cases. This approach is useful to design a mutant protein which differs from the existing one only in a few residues. The approach includes (1) statistical techniques like sequence alignment and secondary structure prediction, and (2) searching techniques which try to find a conformation of the target protein from the structural database that closely resembles the target.

2.2.3 Survey of Molecular Modeling Systems

In recent years, many molecular modeling programs have been developed for different purposes. FRODO [42] and GRASP [58] are molecular modeling programs to compute and display electron-density maps and electrostatic potentials. FRODO is one of the first molecular modeling programs intended for X-ray crystallographers. AMBER [60], CHARMM [8], and GROMOS [73] are molecular modeling and simulation programs including energy minimization, molecular dynamics, free energy perturbation, normal mode analysis, and so on. These packages provide extensive and powerful computational tools and are widely used to simulate molecular structures. WHAT IF [76] is designed for molecular modeling and drug design, providing an environment for displaying, manipulating, and analyzing small molecules, proteins, nucleic acids, and their interactions.

MOLSCRIPT [47] is a molecular drawing program capable of generating schematic drawings [49], wire models, ball-and-stick models, and space-filling models. It creates

molecular graphics in the form of PostScript files. RIBBONS [12] is dedicated to viewing of solid shaded ribbon models of macromolecules. It can create realistic images of various kinds of ribbon models. RASMOL [64] is a molecular graphics program intended for the visualization of proteins, nucleic acids and small molecules.

Recently a couple of systems have been developed using object-oriented technology. NAMD, VMD, and PDBlib are among them. PDBlib [16] is an object-oriented class library written in C++ intended for representing the three-dimensional structure of biological molecules, in particular for PDB (Brookhaven Protein Data Bank) structures. NAMD [23] is an object-oriented molecular dynamics program designed for distributed-memory parallel-processor systems. It uses object-oriented design coupled with a multi-threaded, message-driven computational model. VMD [36, 37] is a molecular graphics and visualization program designed to be used for interactive display of biological molecular systems including visualization of structure and dynamic data such as position, velocity, and energy. MMQL [68] is a query language based on PDBlib, and designed specifically for querying features of macromolecular structure. Besides, several molecular structure databases have been developed using object-oriented database technologies.

Basic features of molecular modeling systems include structure building, visualization and manipulation of three-dimensional molecular models, molecular surface displays, molecular mechanics and dynamics, conformational analysis, and the calculation and visualization of various physical properties such as electron density and electrostatic potential. There are three fundamental areas addressed by molecular modeling software: *computation*, *geometry*, and *graphics*. Computation includes functions to calculate molecular energies and properties associated with energy such as charge distribution, dipole moment, and electrostatic potential. Geometry includes functions to build, manipulate, and measure three-dimensional molecular structures.

Combinations of computational and geometry functions can be used to study problems such as conformational energy surfaces and transitions, reaction pathways, and intermolecular interactions. Graphics functions provide a means of visualizing molecular structure and property data.

The scope of a molecular modeling program is determined by its capabilities in each of these fundamental areas. Specialized programs usually provide functions in a single area, and are used in tandem to achieve a desired goal. For example, structures calculated using an energy minimization program can be displayed using a molecular graphics program. One primary disadvantage in this approach is that the user often has to deal with data compatibility problems between the programs. In addition, using several programs with different styles of interface may be confusing and inefficient.

Multi-purpose programs deliver integrated solutions with a breadth of functionality in order to satisfy the wide-ranging needs involved in diverse areas of research. The goal is to combine all the capabilities in a single package to perform complex operations, and offer the convenience of integrated capabilities under a uniform interface.

In either case, these programs are monolithic where the user has limited flexibility and extensibility. Some users find that such programs are inappropriate or lack capabilities for their needs, needing to add their own functions or customize some portions. Such problems have been partly addressed through general-purpose visualization systems such as AVS [72], IRIS Explorer [25], or Data Explorer [51]. These systems have been originally developed for the general-purpose visualization, and provide extensive tools to analyze and display many types of three-dimensional data. The key advantage of using these systems is that users have the facility to create their own modules which can be easily added to the existing sets.

Recently, several molecular modeling applications [14, 24, 29, 77] have been developed using these environments. However, these systems have some crucial drawbacks for performing general molecular modeling applications. Since these systems are primarily targeted at general visualization applications, they do not provide the data structure which is natural for molecular systems, and are not well suited to serious molecular modeling applications that require extensive computations and structure manipulations. In general these systems must be used in conjunction with other molecular modeling programs.

CHAPTER 3 OVERVIEW OF OBJECT-ORIENTED TOOLBOX FOR MOLECULAR MODELING

3.1 Introduction

Traditional computer codes for scientific applications are mostly monolithic programs designed using conventional programming techniques. Application development generally focused on algorithmic advances of the underlying application, providing rich sets of features but no easy way to add missing features or remove unneeded ones. Today, the rapid changes in the software technologies and environments are fundamentally shifting the way application programs are developed and used. Object-oriented and component software are two primary technical vehicles in modern software development. An increasingly decentralized and communication-focused environment requires a new kind of application that can be rapidly deployed and used. The combination of such new technologies and environments revolutionizes the way applications are developed, distributed, and used. It is highly desirable to design an infrastructure for scientific applications to best exploit these new opportunities.

OTMM (Object-oriented Toolbox for Molecular Modeling) is an object-oriented, component-based application software environment for developing complex molecular modeling applications with interactive visualization and computational functions. The OTMM provides a rich set of reusable components and unified development environments and tools to create a new generation of molecular modeling software. It changes the way developers can create applications by allowing fully functional

applications to be built in the same framework. As an example, a developer could build a special component to support a new simulation algorithm and seamlessly plug it into the OTMM architecture. The OTMM is designed to meet a broad spectrum of requirements:

- Users who want a flexible and easy-to-use application
- Researchers who need to customize their application
- Application developers who want to create specialized components for distribution

The OTMM consists of two primary components: MMFC and Pro3D. MMFC (Molecular Modeling Foundation Class) is the core of the OTMM and contains numerous reusable object classes for building molecular modeling applications or application components. Using MMFC's prebuilt components, programmers can quickly create professional molecular modeling applications and can save a considerable amount of development time. It supports class and component reusability, increasing the ability to develop new applications rapidly from existing ones and incorporate it into existing ones. Researchers can add their own algorithms and operations to meet specific requirements by simply integrating MMFC developed applications as run-time components.

Pro3D is an application program built on the MMFC where custom-built application components can be integrated and run. Two key features that distinguish Pro3D from the traditional programs are the visual programming and component based environments, which make Pro3D more flexible and extensible, easier to use than traditional systems. Pro3D combined with MMFC allows users to create and run a new component rapidly and easily, providing an object-oriented environment for the development and implementation of customized modular toolkits as an alternative to complete packages.

The following sections describe the architecture of MMFC and Pro3D in some detail, introducing their new technologies.

3.2 Molecular Modeling Foundation Class: MMFC

Scientific software systems have usually evolved over years of programming by a number of different individuals, where different programmers typically use different styles and conventions, which leads to inconsistency in the overall software structure. As their feature sets have grown in complexity and size, they have become increasingly difficult and costly to maintain and upgrade. In addition, users of scientific software systems often want to add their own functions or customize some portions.

An object-oriented framework addresses the issues associated with scientific software in general. Designed to be used by the computational scientist, the framework predefines the overall structure of the system and provides the components that are common to its application domain so that the application developer can concentrate on the specifics of scientific application. In addition, object and data encapsulation, data abstraction, polymorphism, as well as inheritance help represent and manage the complex macromolecular structure in a much simpler way.

MMFC is a domain-specific framework written in C++ intended for a broad area of applications in molecular modeling. The philosophy of MMFC is to identify representations that solve extensive classes of molecular modeling problems. Users of the MMFC have access to a wealth of fine-grain programming objects that provide a complete development environment in molecular modeling to make it fast and easy to build applications, and allows programmers to focus on the details specific to the application. It takes advantage of such state-of-the-art modern software technologies as object-oriented design and programming, design pattern, and component software to ensure a higher degree of flexibility, extensibility and reusability.

The framework consists of three basic components: Data Model, Iterators, and Algorithms. The Data Model contains objects for hierarchical data structures specifically

designed to model, manipulate and visualize all the types of data found in molecular modeling. Objects in the Data Model are polymorphic, so each object can operate on many different types of data. If an application requires customized data types for application-specific data, they can be easily defined by inheritance and composition.

Applications in molecular modeling involve extensive sets of scientific and mathematical algorithms. Algorithms are most likely to change in the molecular modeling system. Therefore algorithms should be implemented as separate objects to minimize and localize the effect of the change to the framework. The separation of algorithms from data structures makes it easy to add new operations over the object structure without changing the classes defining that object structure. While MMFC provides numerous general algorithms that can be readily used in molecular modeling, the main objective is to provide researchers with flexible tools that can accelerate algorithm development. Instead of providing just a collection of existing algorithms, MMFC attempts to extract commonalities of general algorithms in molecular modeling and decompose them into primitives that are simple and extensible.

Iterators provide the link between the algorithms and data models by specifying the mechanism that algorithms iterate over the data components. Specifying algorithms in terms of iterators rather than in terms of the components directly makes the independence between these two possible. Iterator is a smart pointer generalizing the notion of pointers and encapsulating information about locations in objects. Iterator has the ability to iterate over any components in any container at any abstraction level without knowing what kind of container it is and its internal structure.

By exploiting the power of C++ templates, classes in MMFC provide a standard way of writing code for every combination of iterators and data type in molecular components. Templates are a code reuse feature of the C++ language that allow developers to define a

specific software implementation without yet knowing the type of data that will flow through it. Developers will define the types of data that are specific to their application. Designed to be reusable, extensible and modular, developers are encouraged to use extensive APIs supported by MMFC to add new component which can be incorporated seamlessly within Pro3D.

3.3 Three-Dimensional Protein Analysis Program: Pro3D

Pro3D is a component container program built on the MMFC where custom-built application components can be integrated and run. Pro3D is designed to be flexible and extensible, easy to use to meet the needs in molecular modeling applications. In addition to basic functionality for structure manipulation and visualization, Pro3D provides two new features unique in molecular modeling systems: Visual programming and component based environments.

3.3.1 Visual Programming

Pro3D's visual programming environment makes it easy for users to quickly and interactively create their modeling procedures. The visual programming environment provides an easy-to-use programming interface where the user graphically connects component modules together to build a modeling network. It displays the modeling application in a flowchart-type module network. This network becomes an application which can be saved, reused and modified. Each module is a function that operates on its input data and produces output. At the heart of the visual programming interface is the network editor, which allows developers to construct applications as connected, hierarchical networks of objects.

3.3.2 Component Software

While Pro3D provides numerous pre-built-in functionalities required for manipulating and visualizing molecules, any new functionalities can be easily integrated at run-time in the form of software components. The components built in the OTMM environment will all inter-operate, allowing the integration of components written by different developers without disrupting the operation of a current system. Independent developers will be able to distribute newly developed components through the Internet, thereby providing a pipeline between the Internet and desktop applications. This approach enables collaboration between scientists in different locations with different areas of expertise. Visual programming coupled with object-oriented and component software allow users to assemble programs with exactly the functionality they need.

3.3.3 Structure Viewer

Another important feature of Pro3D is a powerful and interactive 3-D Structure Viewer that provides a unified display system for viewing the three-dimensional structure of molecules. Structure Viewer provides standard capabilities for visualization of biomolecules which include the abilities to rotate, translate and scale the object in real time with a virtual trackball. Users have control over objects including:

- Choice of rendering mode and color
- Cameras and lights such as separate camera per structure with multiple objects and lights, control of camera orientation and projection, depth cueing
- 3D geometries property support for color, material type, specular color, shading, advanced rendering features such as transparency and texture mapping
- Interactions such as object selections and transformation

CHAPTER 4

MOLECULAR MODELING FOUNDATION CLASS

4.1 Introduction

In this chapter, we discuss the features and architecture of MMFC in detail. A framework is a general architecture for the domain and its classes describe the objects in the domain and how they interact. The single most important process in a framework design is to capture the most general key abstractions in the domain that compose the top level of the class hierarchy. Most applications in molecular modeling can be considered as *algorithms iterated on data components*. For example, in molecular mechanics, force fields are algorithms iterated on molecular components such as bonds, angles, torsions and so on. Here *iteration* means the mechanism by which the algorithms traverse the molecular components. This results in three key abstractions: *algorithms*, *iterators*, and *data components* as shown in Figure 4.1.

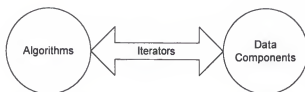


Figure 4.1. The three key abstractions of OFMM

The central part of the framework design lies in the identifying classes and designing interfaces that abstract key objects: Data Model, Iterators, and Algorithms. In addition to these classes, MMFC contains a set of classes to facilitate component-based application

development. These classes will be discussed separately in the next chapter. In the following sections, we present the detailed design of the each component.

4.2 Data Model

Molecular modeling systems consist of complex data structures with many different kinds of data. All aspects of the data structures required by molecular models are stored within a set of object classes known as the Data Model. These classes provide abstract data types (ADT) with publicly defined abstract meaning that is separated from the way the objects are represented and the operations are implemented.

The key steps in designing Data Model classes are:

1. Identify the classes/objects required to model the system.
2. Identify relationships between classes - generalization/specialization, aggregation, and association - by analyzing how the objects collaborate with one another to fulfill a responsibility.
3. Identify member functions and member data required in each class identified in previous steps.

The output of the first step is a number of independent classes responsible for representing molecular structures. The second step makes relationships among these classes and produces a class hierarchy and object relationships diagrams. Finally, each class is implemented in detail with member functions and data in the last step.

Data model of MMFC consists of two basic components: the Molecular Component and Geometry Component. Molecular component classes are responsible for the representation and manipulation of the molecular structure such as atom, bond and residue. Geometry components represent geometrical primitives such as dots, lines and spheres translated from molecular structures. They are used to display the structure and

property of a molecule. Customized components are usually extended by inheriting from the base classes.

4.2.1 Molecular Components

Macromolecular structures consist of a number of various components including polypeptide chains or DNA strands, secondary structures, functional sites, residues, atoms, bonds, angles, torsions and so on. The structure of a macromolecule is hierarchical in nature. For example, a protein molecule may contain one or more chains. Each chain is made up of a series of amino acid residues, and a residue is composed of several types of atoms, as shown in Figure 4.2.

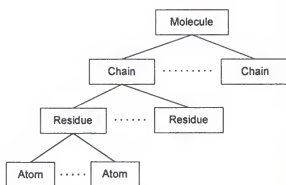


Figure 4.2. Hierarchical structure of macromolecule

Atoms are the fundamental organizational units of the molecular structure. Each atom incorporates various information including atomic coordinates and atomic properties such as name, types, radius, color, and charge. Atoms make up bonds, angles and dihedrals. Atoms are also grouped into substructures, such as subentities and secondary structure. Secondary structures are distinct segments having characteristic shapes that can be classified into helix, turn, and sheet. A subentity may be subdivided into groups, which contain several atoms whose total charge is neutral or a unit charge. This subdivision can be used in the more efficient calculation of nonbonded interactions.

Bond, Angle, and Torsion are key elements for the conformation of a molecule. They determine both the structure and energy of the molecule. There exist various kinds of bonds including covalent bond, hydrogen bond, non-bond, and disulphide linkages. Covalent bonds represent connectivities between atoms in a molecule. Non-bonds refer to van der Waals terms and the electrostatic terms between all atom pairs. A hydrogen bond arises when a hydrogen atom is covalently bonded to other electronegative atoms such as oxygen or nitrogen. The bonding electron density is partly shifted onto the heavier atom and the hydrogen is left with a significant positive charge. Such a hydrogen has a relatively strong electrostatic attraction for oxygen or nitrogen atoms. Hydrogen bonding interactions play an important role in shaping the structure of aqueous systems.

An important feature required in molecular modeling is the ability to specify a subset of molecules including atoms, subentity, bonds, etc., based on a wide range of properties. MolSet is a collection class to hold subset components. MolSet supports a standard set of membership operations: intersection, union, and difference, which are implemented by the operators $\&$, $+$, and $-$, respectively. These operations are defined over heterogeneous components as well as over homogeneous components. MolSet recognizes its elements at run-time and allows the user to perform these manipulations without a direct knowledge of the kind of elements that a collection contains. For example, the user can write a function that takes the intersection of two subsets with any combination of components that are passed as function arguments as in the code:

```
Intersect(MolSet sa, MolSet sb) {
    MolSet common = sa & sb;
}
```

The operation is not commutative when the component of left side is different from that of the right side, and the return type is determined by left argument. For example,

```
MolSet sa; // set of atoms
MolSet sb; // set of subentities
MolSet sc = sa & sb;
```

```
MolSet sd = sb & sa;
```

In this sample code, subset *sc* contains atoms which belong to both subset *sa* and *sb*, while subset *sd* contains residues to which at least one atom in *sa* belongs, and which belong to *sb*. MolSet supports a very powerful abstraction tool in the design of a visual programming and molecular query system. Examples of how to use MolSet are provided in detail later.

Other important components that are not direct parts of a molecule, but essential in structure generation and energy computation are Topology and Parameter. Topology contains the information necessary to describe bond connectivity, angle, torsion, charge distribution, hydrogen bonding, and internal coordinates. Parameter contains the information necessary to determine the structure and energies, and includes force constants and equilibrium geometries of all possible bonds, angles and torsions. These data are required in order to perform all energy-related operations as well as other structural manipulations.

A collection of topologies and parameters are stored and managed by TopologyMgr and ParamMgr, respectively. There exist many different kinds of topologies and parameters developed by different research groups. Even identical amino acid residues may be expressed in different ways. For example, for large systems hydrogens are combined with neighboring heavy atoms to reduce the problem size which is referred to as the extended-atom representation. Having TopologyMgr and ParamMgr take care of those varieties, the overall structure remains unchanged and a single model can accommodate any kind of residue formats.

We can classify these various components into three categories:

- Base components: components directly responsible of comprising structure hierarchy
- atom, subentity, entity, molecule.

- Derived components: components derived from base components - bond, angle, torsion, and group are derived from atoms; secondary structure is derived from subentity.
- Support components: components not a part of structure composition, but essential in structure generation and energy computation - molset, topology, parameter, coordinate.

Having identified various data types, the next step is to design a class hierarchy that efficiently represents the overall structure and relationships among components. The macromolecular structure can be represented at different abstraction levels: molecule, chain, secondary structure, residue, atom and so on. It is important to treat all objects in the hierarchy uniformly so that users don't have to distinguish these objects. The *Composite* pattern uses recursive composition for this purpose. The key to this pattern is an abstract class *MolComponent* that represents both primitives and their containers. The subclasses derived directly from *MolComponent* define primitive molecular components. *MolComponent* defines some basic methods that are common to all the molecular components, which include setting and retrieving name, id, attribute, and parent. Every molecular component has its name and unique id. There may exist multiple components with the same name. Id is used internally in the framework to identify a unique object. Attribute represents the current status of the components such as selection, rendering model and so on. *MolComponents* maintain references to their parents in order to simplify the traversal and management of a composite structure. Providing parent access method simplifies moving up the structure and deleting a component. The simplified declaration of *MolComponent* is shown below.

```
class MolComponent {
public:
    const char* Name() const {return _name;}
    int Id() const {return _id;}
    virtual void SetId();
    unsigned Attribute() const {return _attr;}
```

```

MolComponent* Parent() const {return _parent;}
void SetParent(MolComponent* p) {_parent=p;}

private:
    char* _name;
    int _id;
    unsigned _attr;
    MolComponent* _parent;
}

```

MolComposite is an abstract subclass derived from *MolComponent* that abstracts all aggregate molecular components. The subclasses derived from *MolComposite* define aggregate objects that act as containers for primitives or smaller containers. All container classes support child management functions such as *Add* and *Remove*. This class uses a generic container class such as *list* or *array* to hold and manage its children. The choice of which class to use depends on the context. *Array* has low overhead in terms of storage and ensures constant access time regardless of the size of elements. However, inserting or deleting an element of the array is inefficient. In contrast, *list* can insert or delete an element quickly, but it takes linear access time and the storage overhead is high. Consequently, *array* is used if efficient random access operation is higher in priority, *list* is used if efficient insertion/deletion operation is higher in priority.

Once these two important base classes are defined, all the base and derived molecular component classes are derived from either *MolComponent* or *MolComposite*. Support components are not involved in structure hierarchy and are not derived from these classes. The categorization of all the molecular components are summarized in Table 1 and their class hierarchy is shown in Figure 4.3.

	MolComponent	MolComposite
Base components	Atom	Subentity, Entity, Compound
Derived components	Bond, Angle, Torsion	Group, SecStructure, Molset
Support components	Topology, Parameter, Coordinate	

Table 1. Categorization of molecular components

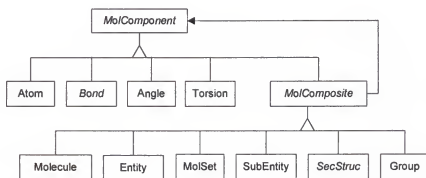


Figure 4.3. Molecular component class diagram

4.2.2 Objects Relationships and Collaborations

A number of different types of objects collaborate to perform a task requested by a user. Molecular component objects are interrelated so that various structural manipulations and computational operations can be efficiently performed. Their relationships and responsibilities can be best described by the use of design patterns. A conceptual representation of the relationships among molecular component classes are shown in Figure 4.4. An arrow with filled circle at one end indicates aggregation relationships. A simple arrow indicates that an object has a reference or pointer to another object.

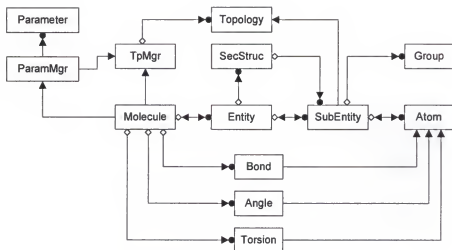


Figure 4.4. Object diagram

Various types of design patterns can be recognized from the relationships of objects. Composite pattern for the hierarchical structure of a macromolecule is represented by a number of aggregation relationships. Bi-directional arrows indicate that the relationship can be established in either direction. This simplifies the traversal and management of a composite structure. All the molecular components are always accessed by the Iterator pattern, which will be discussed in detail in the next section.

TopologyMgr and ParamMgr are examples of the Strategy pattern. Strategy encapsulates a family of related algorithms, and makes them interchangeable without affecting the clients that use them. Topology defines the internal structure of a molecule and many different implementations of topology exist. TopologyMgr encapsulates the variations and lets the molecular structure vary independently of framework. ParamMgr takes care of the variations of different parameters used by different force-fields. TopologyMgr and ParamMgr are also the Singleton pattern, which ensures a class has only one instance, and provides a global point of access to it. Only one instance of TopologyMgr and ParamMgr of the same class is needed. Users may instantiate them each time new structures are created.

4.2.3. Structure Generation and Modification

The first step in molecular modeling is to construct the three-dimensional structure of a molecule and transform it into the desired conformation. Three-dimensional structures can be built either from scratch or from existing structure data files such as PDB (Brookhaven Protein Data Bank). Model building techniques are used to create three-dimensional structures of molecules from scratch. Small molecules such as peptides, saccharides, and nucleotides are amenable to model building. Biopolymers are built by linking together sub-structures from topology and parameter dictionary which are experimentally determined. These lists are usually stored in separate files, and are taken together when generating a structure. The input consists of the sequence of residues and the desired conformation such as alpha-helix and beta-turn. After the structure has been created, structure editing may be required to modify the structures into a new one. Such structure generation and modification are key tools in molecular modeling.

The procedure to build a complete structure consists of several steps. First, the primary structure is specified from a list of residue types. Corresponding residue topologies are taken and all the structural components such as bonds, angles, torsions, and internal coordinates are created from the corresponding information in topology. If connectivity information is not contained in the topology file, bonds are calculated from interatomic distances. Inter-residue bond prescribes which atoms will be selected from the previous or next residue when covalently linking residues together. If information on the angles and/or torsions is not listed in the topology, all possible angles and dihedral angles are generated automatically. The list of bonds, angles and torsions in internal coordinates are filled with the corresponding entries in parameter information. The number of each atom type is used as an index to the parameter. Finally, Cartesian coordinates are generated from the internal coordinates, which represent the positions of

the atoms in the real space. The complete structure holds information on atoms, bonds, angles, torsions, and improper torsions as well as information needed to generate the hydrogen bonds and the non-bonds list, which is essential for the calculation of the energy of the system.

For molecular modeling, two different coordinate systems are used: the *Cartesian coordinates* and the *internal coordinates*. The Cartesian coordinates are given as a triplet for each atom (x, y, z) and represent the positions of the atoms in real space. A Cartesian coordinate can be considered as a three-dimensional vector and represented by Vector class. Vector class provides all the standard vector operations such as dot product, cross product, and norm, as well as operations for coordinate manipulations such as translation, rotation, and scaling.

The internal coordinates consist of values describing the molecular structure that include bond lengths, bond angles and dihedral angles. InternalCoordinate class has methods that convert between Cartesian coordinates and internal coordinates. Given the internal coordinates of four atoms (i, j, k, l) and Cartesian coordinates of three atoms (i, j, k) as shown in Figure 4.5, the position of atom (l), i.e., the Cartesian coordinate, can be obtained by

$$\mathbf{r}_l = \mathbf{r}_k + a\mathbf{r}_{kj} + b(\mathbf{r}_{kj} \times \mathbf{r}_{ji} \times \mathbf{r}_{kj}) + c(\mathbf{r}_{ji} \times \mathbf{r}_{kj})$$

where $a = r \cos \theta$, $b = r \sin \theta \cos \phi$, $c = r \sin \theta \sin \phi$, $r = |\mathbf{r}|$.

In reverse, given Cartesian coordinates of four atoms (i, j, k, l), the bond angle θ formed by atom (i, j, k) and the torsion angle ϕ formed by atom (i, j, k, l) is obtained by

$$\cos \theta = -\frac{\mathbf{r}_{jk} \cdot \mathbf{r}_{ij}}{r_{jk} r_{ij}} \quad \cos \phi = \frac{(\mathbf{r}_{jk} \times \mathbf{r}_{ij}) \cdot (\mathbf{r}_{kl} \times \mathbf{r}_{jk})}{r_{ij} r_{jk}^2 r_{kl} \sin \theta_{jk} \sin \theta_{kl}}$$

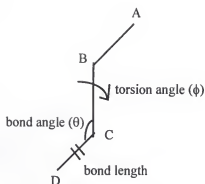


Figure 4.5. Definition of internal coordinate

Structure editing can be used to modify known molecular structures. Experimentally determined or modeled molecular structures can be transformed into new structures using structure editing techniques. Such techniques include:

1. Changing atomic composition such as replacing or deleting atoms.
2. Adding and removing small chemical groups.
3. Changing valence geometry such as bond length or angle.

The major disadvantage of a specialized data structure for biopolymers is that the scope of the Topology and Parameter dictionary-based data determines the kinds of molecules that can be recognized. This can constrain the applicability of atom-by-atom modifications such as a residue with deleted atoms that result in non-standard structures. A non-standard residue is also required to represent the first and last residue of a protein sequence, which is termed N-terminal and C-terminal, respectively. One way to handle nonstandard residues is to add topology data for non-standard monomers to the dictionary. With this approach a new topology is required for every different kind of non-standard residue, resulting in unnecessary overhead. For example, different topologies for N-terminal and C-terminal are required for every kind of amino-acid topology in spite of the fact that the same modification rule is applied. It is more efficient to specify what is to be modified from the topology rather than to keep a modified topology.

MMFC provides this facility through a PatchTopology class inherited from Topology. PatchTopology can alter a standard residue structure or form covalent links between residues by adding responsibilities to standard Topology objects. PatchTopology is an example of the Decorator pattern. Decorators provide a flexible mechanism for attaching additional functionality to individual objects dynamically and transparently. There are three basic operations supported by PatchTopology: insert, delete, and replace. These operations specify what to be inserted, deleted or replaced from a standard Topology when creating a residue structure. If one or more atoms are deleted from a residue, all references to the deleted atoms such as bonds, angles, and torsions should also be deleted from a molecule. This means that the entire structure should be recreated once a change has been made.

4.3 Iterator

Iterators provide the link between the algorithms and data model. An aggregate object such as MolComposite needs to be iterated by the user without knowing the specific kind of container and its implementation. Iterator has the ability to iterate over any container in the hierarchy at any abstraction level without knowing what kind of container it is and its internal structure. Iterators generalize the notion of pointers, encapsulating information about locations in objects. Iterator lets the user traverse all elements of a container transparent to the specific implementation of the data structure.

Most algorithms in molecular modeling work by iterating various molecular components. The key importance of Iterator is that it provides a uniform way for algorithms to access the elements of aggregate objects. The hierarchical structure of a macromolecule is organized such that a position in the structure may span many levels of nested aggregate. For example, to iterate atoms in a protein it is necessary to jump to the next residue when all atoms in the current residue are traversed. By using Iterator to

access molecular components, the interface between molecular components and algorithms can be consistent and independent of the internal representation of data structures.

A basic set of operations that Iterator should support include: dereferencing, traversal, and IsDone. An object in the current location is returned by dereference operator `*()`. Traversal operations include increment, decrement, and random access, which are implemented by operator `++`, operator `--`, and operator `[]`, respectively. To support random access iterators in constant access time, MolComposite class must use Array to store its element. Random access iterators may take containers with List class, but the operation executes in linear time.

OFMM provides iterators to access all the standard types of molecular components such as AtomIterator, SubEntityIterator, BondIterator and so on. All concrete Iterators are derived from abstract Iterator class, which is a template class parameterized based on the component type that defines the interface of all the subclasses.

```
template <class T>
class Iterator
{
public:
    operator bool() {return _current;}
    T* operator *() const {return (T*)*_current;}

protected:
    ArrayIterator _current;
};
```

AtomIterator is a concrete class for iterating atoms:

```
class AtomIterator : public Iterator<ATOM> {
public:
    AtomIterator(const Compound& c);
    AtomIterator(const Entity& e);
    AtomIterator(const SubEntity& se);
    AtomIterator(const MolSet& ms);
    AtomIterator& operator++();
    AtomIterator& operator--();

private:
    SubEntityIterator _seIter;
};
```

It has several constructors to accommodate all the types of components over which atoms can be iterated. `operator++` advances the position to the next atom. `AtomIterator` uses `SubEntityIterator` to advance to the next residue if the current position is the last atom of a residue.

```
AtomIterator& AtomIterator::operator++() {
    ++_current;
    if (!_current) { // end of this residue, advance to next residue
        ++_seIter;
        if (_seIter) {
            _current = (*_seIter)->_childs.begin();
        }
    }
    return *this;
}
```

`AtomIterator` is used to retrieve every atom in a compound without worrying about its internal representation.

```
for (AtomIterator i(*compound); i; i++)
    char* name = (*i)->Name();
```

`Filter` is an advanced type of `Iterator`. It has the same interface as `Iterator`, but iterates only over those elements that satisfy a specific condition. Internally `Filter` uses `Iterator` to do the traversal.

```
template <class MCIter, class T>
class NameFilter
{
public:
    NameFilter(const MCIter& iter, const char* name)
        : iter(iter), _name(name) {
        while (iter && (*iter)->Name() != _name)
            ++iter;
    }

    operator bool () const {return iter;}
    T* operator *() const {return *_iter;}
    MCIter& operator++() {
        ++iter;
        while (iter && (*iter)->Name() != _name)
            ++iter;
        return iter;
    }

private:
    MCIter iter;
    char* _name;
};
```

We can define different filters for various types of components by parameterizing the

NameFilter:

```
typedef NameFilter<EntityIterator, Entity> EntityFilter;
typedef NameFilter<SEIterator, SubEntity> SEFilter;
typedef NameFilter<AtomIterator, Atom> AtomFilter;
```

4.4 Algorithm

Applications in molecular modeling involve extensive sets of scientific and mathematical functions. Typical design approach encapsulates data structures as objects and algorithms as member functions in the objects. With this approach, objects that depend on an algorithm will have to change when the algorithm changes. Algorithms are most likely to change among others in the molecular modeling system. Scientists will need to extend and customize the existing algorithms as well as add their own algorithms to apply to a specific system. Therefore algorithms should be implemented as separate objects to minimize and localize the effect of the change to the framework [55, 56]. Separating algorithms from components on which they operate provides several advantages. If algorithms are associated with particular classes, an algorithm may be added by either adding a member function to the classes or deriving subclasses. Both approaches affect all the component classes associated with the algorithm and cause difficulty whenever new algorithms are added. Another benefit is that algorithms and the object structure can be designed and evolve independently. The characteristic of algorithms is quite different from that of molecular components and each class should be designed based on different criteria. Classes for molecular components compose a large class hierarchy, and are usually extended and customized by inheritance. In contrast, algorithms consist of fine-grained, interchangeable building blocks that allow many useful combinations to build new larger algorithms. Generality can be achieved without

sacrificing efficiency by avoiding complex inheritance and extensive use of virtual functions.

It is important to capture the set of common functionalities that can serve as fundamentals for all the applications in the domain for maximum reuse and flexibility. The strategy in designing MMFC algorithm classes is not to provide a number of specific functions but to provide a general and flexible mechanism that can be easily adjusted to meet user's requirements in any situation. For this purpose, a large algorithm is decomposed into small algorithmic building blocks, which are implemented as separate classes. These components provide functional primitives at a higher level of abstraction that can be shared by several functions. New larger algorithms are obtained by composing these objects rather than inheritance. Two important components responsible for the design of algorithms are: function object and template function.

Small functional components are implemented as function objects. A function object is an object of some class type that provides the `operator()()` member function that enables the use of the class object as if it were a function. Function objects are fine-grained, interchangeable algorithmic building blocks implemented as separate classes from which larger algorithms can be composed, and provide reusable solutions to many different problems. It can be passed as an argument to other functions to compose a larger algorithm. Passing a function object as an argument provides an important advantage over passing a pointer to a function. It is efficient because it can be done at compile time and can be expanded as an inline function. This feature significantly enhances the performance of highly iterative algorithms.

While function objects define primitive operations on each component, template functions provide the mechanism for a function object to iterate over aggregate components. The reason for using template functions is that algorithms are isolated from

the specific implementations of data structures and parameterized by function objects and Iterators. A typical template function takes a function object and Iterator as arguments, which looks like:

```
template <class MCIter, class Function>
double Accumulate(MCIter _iter, Function _function);
```

To demonstrate the composition of an algorithm using function objects and template functions, consider a mathematical expression $\sum_{\text{components}} f$, which is one of the most common operations in molecular simulation methods. This expression corresponds to computing a function f on each molecular component and then accumulating the result as the components are traversed. This operation can be implemented as `Accumulate(Iterator(component), f)`, where `Accumulate` is a template function and f is a function object. Iterators in conjunction with function objects provide a mechanism to abstract the complex control flow including a loop and iterating the structure of a component. The function object f can be any user-defined function. The writer of the function f need only to write the actual function body without being concerned about how to retrieve components on which the function operates and how this function is to be called in the main procedure.

The complexity of algorithms varies greatly. An example of a simple algorithm may be collecting the coordinates of backbone atoms to generate a display model with line structure. More complex algorithms include energy minimization and molecular dynamics simulation. There exist three categories of key algorithms that encapsulate the domain:

- Generator function: algorithms that create new molecular components or geometry components.

- Predicate function: algorithms responsible for selecting a subset of components based on various criteria.
- Mathematical function: algorithms that involve general mathematical functions, which make changes to components' internal state or return numeric values.

A wide range of functions in molecular modeling can be implemented using the standard model: a Template function parameterized based on Iterator and Function object.

The following sections present the detailed design methods in each category.

4.4.1 Generator

This class of algorithms is responsible for creating new components from information of existing components. Generators are divided into two groups: MCGenerator and GCGenerator. MCGenerator creates new molecular components such as bonds and angles from the existing structure. They are mostly used in structure building and energy computation. GCGenerator generates the geometry components such as line segments or vertices which are used to display the structure and property of a molecule.

MCGenerator. Information on bond, angle, and torsion is usually provided by topology, however, it is sometimes necessary to create all the possible bonds, angles, and torsions without explicit information from topology. Bonds are created from atomic coordinates. If a distance between two atoms is within the specific range, two atoms are regarded as covalently bonded. Angles are created from bonds, and torsions are created from angles.

Nonbonded interactions refer to van der Waals terms and the electrostatic terms between all atom pairs. The total number of terms in the potential energy function is dominated by the number of nonbonded interactions. For efficient calculation of nonbonded energy terms, a list is created which contains all nonbonded interactions to be considered. Some pairs of atoms are excluded from the nonbond lists because their

interactions are described by other terms. Directly bonded atoms and the 1-3 atoms of an angle are usually excluded from the nonbond calculation. Hydrogen bonds, and dihedral 1-4 interactions are excluded optionally. Nonbond list contains all atom pairs that are within a predefined cutoff distance unless they are included in a nonbond exclusion list. Since the interatomic distance will change while the molecular structure undergoes computational procedures, the list is updated periodically.

A straightforward algorithm for nonbond generation examines every pair of atoms as:

```
loop over every pair of atoms
  if distance between two atoms < cutoff
    add this pair to nonbond list
  end loop
```

This algorithm would require $O(N^2)$ complexity where N is the number of atoms in the system because there can be up to $N(N-1)/2$ nonbonded interaction pairs. For large systems containing thousands of atoms, this approach may take significant amount of computing time.

To reduce the complexity of nonbond calculation groups are introduced. A group contains several atoms whose total charge is neutral or a unit. Distance calculations are performed only for those atoms whose corresponding groups are in contact. Groups are defined as being in contact if the closest distance between the minimum-sized rectangular boxes containing each of the groups is less than the nonbond cutoff distance. The following pseudo-code shows the procedure for generating a nonbond list by groups:

```
loop over groups
  calculate the center of geometry of each group
  calculate a rectangular box surrounding the group
end loop
```

```

loop over pair of groups
  calculate the minimum distance  $d$  between two rectangular boxes
  if  $d < \text{cutoff}$ 
    loop over pair of atoms from each group
      if the distance between two atoms  $< \text{cutoff}$ 
        add this pair to nonbond list
    end loop
  end loop
end loop

```

This algorithm is still the order of N^2 , however skipping the inner loop by the cutoff distance checking between groups has enhanced the speed 10-fold.

GCGenerator. To visualize the structures and properties of a molecule using various types of display models such as line, ribbon, ball and stick, and space filling, the molecular components are translated to geometric components such as points, lines, splines, spheres, polygons, tubes, texts, and surfaces. Once the geometric primitives have been defined, they are rendered with the appropriate rendering parameters including position, coloring, illumination, and surface properties. The algorithms in GCGenerator class create these geometric primitives that can be rendered on the screen from the structure of a molecule.

These classes of algorithms use either spatial or property information from molecular components such as atoms or bonds. For convenience, algorithms in GCGenerators are divided into two categories: simple algorithms and complex algorithms. In simple algorithms, geometric components are created directly from the molecular components with one-to-one correspondence without further processing. Generating lines or cylinders from bonds, and spheres from atoms belong to this category. A template function SimpleGCGen iterates over the molecular components and generates a set of geometric components:

```

template <class MCIter, class Function>
GeomSet* SimpleGCGen(MCIterator iter, Function MctoGC) {
    GeomSet* root = new GeomSet;
    while (iter) {
        root->Add(MctoGC(**iter));
        ++iter;
    }
    return root;
}

```

An object function MctoGC performs an actual translation from a molecular component to a geometric component. Since SimpleGCGen is parameterized base on the Iterator type and translate function, the same function can be used to generate any kind of geometric components as long as it is a simple algorithm. SphereAtom is an example of a function object that translates the position of an atom to a van der Waals sphere. It has only one public method: `operator()()`.

```

class SphereAtom {
public:
    GeomComponent* operator()(const Atom& atom) const
    {return new Sphere(atom.Point(), atom.Radius());}
};

```

The user generates a space filling model that can be rendered on the screen:

```
GeomSet* model = SimpleGCGen(AtomIterator(compound), SphereAtom);
```

The same information can be represented in several different ways. For example, a bond can be drawn as a simple two-dimensional line which connects two atoms in the bond or be rendered as three-dimensional solid tube. In this case, the user writes two function objects:

```

class LineBond {
public:
    GeomComponent* operator()(const Bond& bond) const
    {return new LineSegment(bond.BeginPoint(), bond.EndPoint());}
};

class SolidBond {
public:
    GeomComponent* operator()(const Bond& bond) const
    {return new Cylinder(bond.BeginPoint(), bond.EndPoint());}
};

```

To generate a 2D line model:

```
GeomSet* 2dLine = SimpleGCGen(BondIterator(compound), LineBond);
```

To generate a solid line model:

```
GeomSet* solidLine=SimpleGCGen(BondIterator(compound), SolidBond);
```

In complex GCGenerator algorithms, there is no one-to-one match between molecular components and geometric components. Generating a ribbon or solvent accessible surface (SAS) belongs to this category. These algorithms usually generate many geometric components from one or more molecular components. The standard form of the template function which is parameterized by Iterator and function object cannot be used since information from more than one component may be necessary at a time.

4.4.2 Predicate

An important feature required in molecular modeling is the ability to specify a subset of molecules based on a wide range of properties. The features of macromolecules are expressed by various types of patterns. Some important patterns are:

Sequence Pattern (Atom, SubEntity, Entity): This pattern type defines combinations of residue or atom types in sequence. The pattern can be either names such as ALA-SER-ALA, or numbers. This pattern also supports wildcard characters: * (match any string), % (match a single character), # (match any number), + (match any digit).

Secondary Structure Pattern: A secondary structure pattern defines a search for a sequentially ordered set of subentities belonging to a predefined type of conformation. There are three types of secondary structure with proteins: helix, sheet, and turn.

Property Pattern (Atom, SubEntity): This type of pattern is defined based on the physical properties of residues or atoms such as volume, polarity, charge, temperatures, B-factor, occupancy, hydrophobicity, and exposure. Some properties such as hydrophobic specify fixed types of residues and define a global set.

Contact Pattern(Atom, SubEntity): This pattern type examines the spatial relationship between collections of residues or atoms based on a variety of criteria such as at (coordinate), around at, nearest to geometric center, inside cylinder, inside box.

Conformational Pattern: A conformational pattern defines a search for a conformation described in terms of coordinates. The coordinates can be either Cartesian or internal.

Covalent bonding pattern: All the atoms that are bonded to at least one other atom, connected (at, from:to).

Surface pattern: The properties which can be used for surfaces are coordinates, screen position, potential, distance, curvature, assigned color and vertex number. For atoms, charge, radius, molecule number, as well as atom name, residue number, residue name, chain name, and accessible area in addition to those of surface properties.

Filter provides the ability to iterate only over those elements that satisfy particular patterns. Filter is parameterized by Iterator and Predicate. The interfaces of Filter are similar to those of Iterator.

```
template <class MCIter, class Predicate>
class Filter
{
public:
    Filter(const MCIter& iter, const Predicate& f)
        : _iter(iter), _compare(f)
        {while (_iter && !_compare(*_iter)) ++_iter;}

    operator bool () const {return _iter;}
    MCIter& operator *() {return _iter;}
    MCIter& operator++() {
        ++(_iter);
        while (_iter && !_compare(*_iter))
            ++_iter;
        return _iter;
    }

private:
    MCIter _iter;
    Predicate _compare;
};
```

Predicate is a function object that determines whether a component satisfies a specific pattern. An example of contact patterns that determines whether an atom is within a specific distance from the reference atom is:

```
class IsWithin {
public:
    IsWithin(Atom* ref, float dist)
        : _reference(ref), _distance(dist) {}
    bool operator()(const Atom& atom) const;
private:
    Atom* _reference;
    float _distance;
};

bool IsWithin::operator()(const Atom& atom) const {
    Vector v1 = atom.Point();
    Vector v2 = _reference->Point();
    return Length(v1-v2) < _distance;
}
```

Users use Filter to retrieve those atoms that are within 5 Å from a reference atom.

```
Compound cmpd;
Atom* ref;
for (Filter f(AtomIterator(cmpd), IsWithin(ref, 5)); f; f++)
    char* name = (*f)->Name();
```

This technique allows users to implement any kind of custom pattern by simply writing a corresponding Predicate function object. We will present more general biological query methods based on this technique in Chapter 6.

4.4.3 Mathematical Functions

The difficulty with designing reusable components for various mathematical algorithms required in molecular modeling is that there exist so many variations associated with the process of computation. For example, the set of force-field functions contains adjustable parameters to obtain the best result depending on the problem. Furthermore, there exist many different kinds of force-fields whose mathematical form varies in each implementation. Some include additional terms to improve the accuracy of the mechanical model. Molecular simulation methods sometimes run with constraints on the structure to enhance the performance. Even if the framework could support all the

existing functions with all the possible variations, new requirements will soon arise and existing codes may not be fully reusable.

The main objective is to provide researchers with flexible tools that can accelerate algorithm development instead of providing just a collection of existing algorithms. For this purpose, MMFC attempts to extract commonalities of general algorithms in molecular modeling and decompose them into primitives that are simple and extensible. Developers only have to write a simple function object that implements a corresponding mathematical equation. The framework takes care of the rest. BondEnergy is an example of a mathematical function object that computes the energy of a single bond.

```
class BondEnergy {
public:
    double operator()(const Bond& bond) const;
private:
    ParamMgr _pMgr;
};

double BondEnergy::operator()(const Bond& bond) const {
    float F = _pMgr->Force(bond);
    float S = _pMgr->Value(bond);
    double s = bond.Length();
    double db = s - S;
    return F * db * db;    // Energy = F * (s - s0)**2
}
```

Accumulate is implemented as a template function parameterized based on Iterator type and function object:

```
template <class MCIter, class Function>
double Accumulate(MCIter iter, Function function) {
    double sum = 0;
    while (iter) {
        sum += _function(**iter);
        ++iter;
    }
    return sum;
}
```

Total bond energy of a compound can be obtained by

```
double energy = Accumulate(BondIterator(compound), BondEnergy);
```

Using function objects in the design of mathematical algorithms provides some other important benefits. For numerical programming, use of first-class functions results in

code that is more elegant and reusable. In functional languages, functions are the first-class objects that may be passed to and returned from other functions. Though C++ is not quite a functional language, it is possible to simulate such features using a function object that can be viewed as a part of the programming language.

There is often a case in which a function is applied to the result of another function, or a function uses the internal results of another function that is computationally expensive. Compose function allows a function to be applied to the result of another function. For example, if $f(x) = 2x$ and $g(x) = x + 2$, $\text{Compose}(f, g)$ is equivalent to $f(g(x)) = 2(x + 2)$. Another case is that in which the result of function application may be another function, that is a function can return another function instead of a simple value. An example is using cutoff functions in the calculation of nonbonded interactions. In the computation of nonbonded interactions, various cutoff functions are used to reduce the number of interacting pairs and hence the time of computation. A function object `CutoffFunction` takes the distance of an atom pair as an argument and returns an appropriate cutoff function object depending on the distance, which is applied to another function object such as `Energy`. This example illustrates key advantages using function objects as first-class like functions. The main procedure that actually computes the energy and/or gradient does not have to allow for the type of cutoff function. The implementation of the cutoff function can be changed without affecting any other functions. Examples of more complicated applications that make extensive use of function objects are discussed in a later chapter.

CHAPTER 5

APPLICATION DESIGN AND VISUAL PROGRAMMING

5.1 Introduction

The OTMM provides an open, flexible application development environment that allows developers and users to create and use scientific applications much more easily than using traditional methods. It provides a predefined control and data structure as well as an extensive set of reusable objects that allows the user to concentrate only on application-specific functions. Two distinctive features of OTMM are a component-based application development environment and visual programming environment. The developer can build a customized component to support a new algorithm and seamlessly plug it into the OTMM environment. The components built in the OTMM environment will all interoperate, allowing the integration of components written by different developers without disrupting the operation of a current system.

Application components developed using OTMM can be classified into two categories: internal and external. An internal component is implemented as an object class derived from MMFC classes and bound to the existing application at compile-time. This approach is simple, but the entire body of the application program should be recompiled to upgrade the system. In contrast, an external component is implemented as a separate software component and bound to the existing program at run-time. This method does not disrupt the operations of the existing system and the user only has to insert the new component at run-time to access its functionalities.

Visual programming is an automatic benefit for users of the component-based approach. The visual programming environment provides a novel user interface where the user graphically connects component modules together to build a modeling network. User interface design is becoming increasingly important as software systems become larger and more complex. An interface of the molecular modeling system which handles complex structures and provides extensive functionalities tends to become complicated and difficult to use. Though a menu-based graphical user interface is intuitive and easy to use, some features are not easily implemented in menus. It is hard to structure the menu system with a large number of possible choices and actions involving logical connectives. In addition, such a menu-based system is not suitable for creating a script file for later use. Traditional molecular modeling software systems such as AMBER and CHARMM use a custom command language to supplement complex or repetitive operations not found on the menus. However, the language-based methodology is difficult to use and implement. Designing a language and language processor is no trivial task. Scientific software may be difficult to learn and use, requiring knowledge not only of the domain, but of the software itself. Having to learn a command language may be another burden to users who are not familiar with programming. The visual programming approach addresses these issues with which complex actions can be easily implemented and used without sacrificing generality and completeness.

In this chapter, we extend the features of OTMM to support the application development and visual programming that allows a user to easily build, manipulate, and visualize a complicated structure in complex molecular modeling applications.

5.2 Visual Programming

The idea of implementation by connection, called dataflow programming, has been applied in many areas, particularly scientific visualization systems and environments for

visual programming [25, 51, 72]. Modules are functional building blocks where each module implements a specific function in the modeling process. The modules correspond to software components and can be interconnected to build large applications in the form of a components network. The network connections specify the type and path of data communicated between the modules, implicitly determining the module's execution order. Modules have one or more input and output ports, and parameters are used to control the module's computation. Data is communicated between modules through ports.

A network editor is used to construct the dataflow network from a palette of modules using a visual programming interface. The editor does not simply manipulate icons, but dynamically configures and controls running component networks. As the graph is constructed, instances of corresponding modules are created and connected. Once a network of modules has been constructed, it can be saved on disk for later use, eliminating the need to use a custom script language.

One of the primary tasks involved in dataflow programming is developing the mechanism in which the network of modules is translated and executed. A common procedure is that the network diagram constructed by the user is translated into a script in a dataflow language, which is then interpreted and executed. In OTMM, we have developed a simple and easy method to translate and execute the module network by eliminating all the intermediate processes. This procedure will be discussed in detail below.

Data representation is another significant problem in a visual programming system. It is important that the modules can be connected in a large variety of ways to facilitate interoperability so that each module may operate on any input regardless of its data type. This issue is exactly what polymorphism in object-oriented programming is intended for.

Since all the data models are derived from a common base class in MMFC, this is no longer a problem in our design.

Three base classes responsible for dataflow programming and execution are Module, Port, and Connector. A user-application is implemented as a subclass of Module and they are interconnected through Connector from Output Port to Input Port between modules. Easy extensibility through the addition and connection of user-developed modules to meet unique requirements is one of the most important features of our design. We present the detailed design on each class in the following sections.

5.3 Module, Port and Connector

5.3.1 Module

When designing an application one should first determine:

- The function of the application
- The input that the application will take
- The parameters that the application should take
- The output of the application

Each user application component is implemented as a subclass of Module. The module encapsulates both the application function and user interface. A module has drawing methods that represent the flow network on the user's screen called by the network editor. It has its own input and output dialogues specifically designed for the function of the module. The input dialog defines the parameters used to control the module's computation. A dialog can contain various control widgets such as push buttons, radio buttons, check buttons, text input, and dials. The optional output dialog defines the way in which the application's internal status is displayed to the user. The complexity of a module is scalable depending on the application design. An application may consist of a single module or several modules.

The abstract base class Module defines the interfaces. It has two abstract virtual functions: Execute and Transfer, which should be implemented by concrete derived classes. Execute defines the main body of an application function. Transfer defines the method that pass data through the flow network.

```
class Module {
public:
    virtual bool Execute() = 0;
    virtual bool Transfer() = 0;
    virtual int NumInput() const {return 0;}
    virtual int NumOutput() const {return 0;}
};
```

The module is divided into four major categories: Producer, Consumer, Transformer, and Operator. Each module is represented by an icon in the network editor with input ports along the top edge and output ports along the bottom edges. Different types of modules are distinguished by the configuration of ports as shown in Figure 5.1.

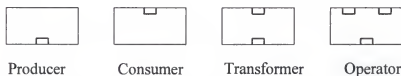


Figure 5.1. Icons representing four different types of modules

Producer is a module which has no input ports. Modules which load a structure file and generate their own data belong to this group. Consumer is a module that only has an input port. Modules that perform computations which only change the internal status or return numeric values fall into this group. Transformer has one input port and one output port. This category includes modules that select a subset of a structure and modules that convert molecular components into geometric components for visualization. The Operator module performs logical operations between two subsets such as union, intersection, and difference. It has two input ports and one output port. The class declarations of Producer and Consumer are shown as:

```

template <class T>
class Producer : public Module {
public:
    virtual bool Transfer() {return _out.Transfer();}
    virtual int NumOutput() const {return _out.NumConnectors();}
    OutPort<T>* GetOutPort() {return &_amp;_out;}

protected:
    OutPort<T> _out;
};

template <class T>
class Consumer : public Module {
public:
    virtual bool Transfer() {return _out.Transfer();}
    virtual int NumInput() const {return _in.NumConnectors();}
    InPort<T>* GetInPort() {return &_amp;_in;}

protected:
    InPort<T> _in;
};

```

5.3.2 Port

Subclasses of Module contain instances of the input and/or output ports depending on the type of the module. Port is a template base class parameterized by the component type. InPort and OutPort are derived from Port, which specialize the input port and output port, respectively. Two component types that parameterize the Port class are: MolSet and GeomSet, which account for molecular components and geometric components, respectively. Each output port has a specific data type it produces and each input port has a specific data type it accepts. When connecting an output port of one module to an input port of another, the data between two ports must be of the same type. Since there exist only two data types, MolSet and GeomSet, the data compatibility between ports is not a big issue here. The Port class is declared as:

```

template <class T>
class Port
{
public:
    void Put(T* set) {_set = set;}
    T* Get() {return &_amp;_set;}

protected:
    T* _set;
};

```

Connections between ports are represented by the Connector class. InPort just keeps the number of connectors attached to it. When a connector is added, the variable `_numConnectors` is incremented. When a connector is removed, it is decremented.

```
template <class T>
class InPort : public Port<T> {
public:
    int NumConnectors() const {return _numConnectors;}
    void Add() {++_numConnectors;}
    void Remove() {--_numConnectors;}

protected:
    int _numConnectors;
};
```

OutPort contains a list that holds instances of the Connector classes attached to it. It delegates Transfer and Remove operations to Connector.

```
template <class T>
class OutPort : public Port<T>
{
public:
    // delete all the connectors in the list _connectors
    virtual ~OutPort();
    void Add(Connector* cn) {_connectors.Add(cn);}
    int NumConnectors() const {return _connectors.size();}
    bool Transfer() {
        for all the connectors c in the list
            c->Transfer();
    }

    void Remove() {
        for all the connectors c in the list
            c->Remove();
    }

private:
    List _connectors;
};
```

5.3.3 Connector

A connector attaches two ports - from InPort to OutPort. The base class Connector is declared as:

```
class Connector {
public:
    virtual ~Connector() {}
    virtual bool Transfer() = 0;
    virtual void Remove() = 0;
};
```

Two concrete subclasses are derived from Connector: MolConnector and
GeomConnector. MolConnector is responsible for connecting ports parameterized by
MolSet. GeomConnector is responsible for connecting ports parameterized by GeomSet.

```
class MolConnector : public Connector {
public:
    MolConnector(OutPort<MolSet>* from, InPort<MolSet>* to);
    virtual ~MolConnector() {_to->Remove();}
    virtual void Transfer();
    virtual void Remove() {_to->Remove();}

private:
    OutPort<MolSet>* _from;
    InPort<MolSet>* _to;
};
```

Connections are established when instances of Connector are created. The constructor of
subclasses of Connector attaches itself to the Ports. When a connection is removed, a
connector detaches itself from the ports before being deleted.

```
MolConnector(OutPort<MolSet>* from, InPort<MolSet>* to)
    : _from(from), _to(to)
{
    _from->Add(this);
    _to->Add();
}
```

Data transfer between modules is actually performed by Connector, which is initiated by
Module and delegated to Connector through Port.

```
void MolConnector::Transfer()
{
    MolSet* ms = _from->Get();
    _to->Put(ms);
}
```

5.3.4 Execution of Module Network

One of the primary tasks involved in dataflow programming is the mechanism in
which the network of modules is translated and executed. Most visual programming
systems that use the dataflow programming method require a number of translation steps
to generate an executable code. We have developed a simple and efficient method that

executes the module network directly from the instances of modules eliminating all the intermediate processes.

Each instance of a user module is stored in a linked list when it is created. The Dispatch function scans the list and retrieves a module with no connector on an input port that comes first in the list. It calls the module's Execute operation, and then transfers the result to all the modules which are connected to the output port of this module. Once the module is successfully executed and data are transferred, the dispatch function removes that module from the list. This process is repeated until the list becomes empty. The order of execution is implicitly determined by the connection of modules. The following pseudo-code describes the dispatch algorithm:

```

loop over the list of modules
  for the first module m in the list with no input
    execute m
    transfer result data to all the modules which are connected to m
    delete all the connectors connected to the output port of m
    delete m from the list
  goto the start of the loop (repeat until the list becomes empty)
end loop

```

An example of a module network is shown in Figure 5.2. Each module has a name that represents the function of the module. The Producer module, named PDBLoad, loads the PDB structure data file and creates a new molecule. This module builds a structure using a topology database either from CHARMM [8] or PDBLib [16]. The created structure is transferred to three Transformer modules that select subsets of a molecule based on various criteria. The bond information in components selected by the first Select module is transformed into lines by Line module. The Operator module AND finds the intersection of two subsets and feeds the result to the Transformer module Sphere which

translates selected atom information into spheres. Finally, the transformed geometric components are transferred to Viewer module to display the desired structure.

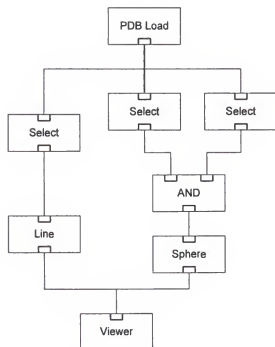


Figure 5.2. An example of a network of modules

The module network is perfectly suited to represent complex procedures occurring in modeling and visualizing molecular structures. It provides the user with an easy-to-use interface as well as the visual flow of a modeling process. In addition, the network can be saved on disk for later use, eliminating the need to use a custom script language.

CHAPTER 6 STRUCTURE MANIPULATION AND VISUALIZATION

6.1 Introduction

The complexity of biological molecular structure necessitates the use of physical or pictorial representations to aid in interpretation, manipulation, and understanding of the structure. Since Raymond Pepinsky first developed an analog machine to transform his x-ray crystallographic data into a molecular picture in 1947, and the first developments in interactive molecular graphics at Project MAC, MIT in 1964, computer graphics have proven to be an indispensable tool for visualizing the three-dimensional structure of molecules [35].

With the introduction of computers with high performance graphic capability, computer graphics have enabled scientists to produce detailed pictures of enormous molecules, and assist the study of the shape of a biological molecule and its interactions with other molecules. Computer graphics can also capture the elusive dynamic behavior of biological molecules. In trillionths of a second, molecules vibrate, twist and rotate. Dynamic simulation can follow the motions of a molecule through thousands of time steps, generating a corresponding number of snapshots of the changing structure. The collection of coordinate sets represents the trajectory of motion and can be played back using computer graphics.

To view the three-dimensional structure of a molecule a number of basic three-dimensional graphic functions are required such as transformation, projection, clipping, and hidden surface-removal. Advanced functions such as lighting, shading and texture

mapping [71] provide enhanced realism and readability. Molecular graphics is concerned with the visualization of three-dimensional molecular structures and properties including display of molecular surfaces, analysis of protein folding and molecular dynamics trajectories, visualization of electron density map and electrostatic potential, and so on.

Visualization software tools for science and research fall into three basic categories: low-level graphics libraries, discipline-specific visualization tools, and general-purpose visualization tools. Graphics libraries such as GL [57], GKS [34], and PHIGS [1] are examples of the first method. The difficulty with this method is that, though these libraries provide collections of low-level graphics operations, they fail to hide the underlying complexity of the visualization problem in molecular graphics and their effective use requires programmers to understand the graphics primitives and data structure inherent in the library. Consequently, they require a professional programmer to develop the program for a specific visualization problem and place a huge software development burden on scientists.

Discipline-specific tools such as RIBBONS [12], MIDAS [27, 28], and O [41] are applications that offer visualization capabilities targeted to a specific discipline in molecular graphics. While these applications provide a large set of built-in functionalities tailored to the discipline, they fail to meet the critical needs of scientists who will want to extend and customize software for their individual needs by adding their own algorithms and tailoring existing ones.

Such problems have been partly addressed through general-purpose visualization systems such as AVS [72] or IRIS Explorer [25]. These systems provide an environment for the development and implementation of customized modular toolkits that meet individual needs. The drawback with these systems for use in molecular graphics is that,

since they are designed for general-purpose use, their abilities to provide data structure and functions specific for molecular modeling are very limited.

We present a new visualization technique specifically designed for molecular graphics using OTMM. The method developed here addresses the problems associated with the existing techniques by providing the common visualization requirements for applications in molecular modeling in an object-oriented environment.

6.2 Molecular Graphics

Molecular graphics is concerned with the graphical representation of three-dimensional molecular structures and properties. Major uses for molecular graphics are

1. As part of the graphical interface of a molecular modeling program, in which the molecular structure drawing can be used to interact with the contents of the molecular data structure (e.g. to insert and remove atoms, or change the valence geometry).
2. For interactive visualization, in which high performance graphics such as depth cueing, perspective, clipping, color coding, and labeling are used to convey the three-dimensional sense of molecular structures. Standard capabilities for visualization include the abilities to rotate, translate and scale the object in real time.
3. As an aid in the analysis of molecular modeling data, in which spatial and structural properties are represented by surfaces, contours, and vectors, and animation can be used to represent time-dependent processes.

The two primary data types involved in molecular graphics are molecular structures and molecular properties.

6.2.1 Visualization of Molecular Structure

Various display models have been developed to efficiently represent molecular structures. Among them are wireframe, ball and stick, backbone trace, ribbon, and space filling models, and dot surface representations. The wireframe model is a collection of

lines that connect every pair of atoms forming a bond. In the ball-and-stick model, bonds are represented as cylinders and a sphere represents each atom. Different atom and bond types may be distinguished by size and color.

With the complexity of macromolecular structures, it is often necessary to create simplified and visually understandable representations. Some of the most useful simplifications are the backbone trace and ribbon model. The backbone trace model is composed of lines that connect the alpha-carbon atoms of a protein. In the ribbon model, the backbone of a protein is interpolated by a smooth ribbon of spline curves instead of straight lines. Two sorts of spline algorithms [43] are usually used to generate smooth curves between backbone atoms: B-spline and Cardinal spline. The use of B-spline produces a very smooth interpolation, however, a B-spline does not in general pass through the control points used to define it. The Cardinal spline will not produce as smooth a ribbon as B-spline, but it will curve to exactly fit through each control point in the backbone. Ribbon model is a simple but efficient graphic representation to visualize the way a protein chain folds into a compact molecule, eliminating the confusing tangle of individual atoms. A schematic diagram based on the work of Richardson [62] is an advanced ribbon model which characterizes the secondary structure, representing alpha-helices as cylinders, beta-sheets as arrows, and tubes for coils and turns. This representation greatly enhances the topological relationships among elements of secondary structure in proteins.

The space-filling model first used by Corey, Pauling and Koltun, hence called the CPK model [15], represents the volume enclosed with various parameters by a shaded sphere. These parameters include the van der Waals radius, solvent accessible surface, and electrostatic potential. The CPK model aids in visualizing the spatial distribution and

conformation of molecular structures and enables enhanced perception of critical biological interactions commonly occurring at this surface.

The dot sphere model is different from the space-filling model in that the surface of a sphere is represented by uniformly distributed points rather than a solid surface. This representation of a sphere can be rendered more efficiently than solid modeling since time-consuming shading and hidden-surface removal algorithms can be avoided. Another advantage with this model is that the dot representation of a sphere results in transparent surfaces and allows the user to mix other models with the space-filling features. Protein interiors can also be viewed using this model by the cutting of serial sections through a protein.

6.2.2 Visualization of Molecular Properties

The concept of molecular surfaces is very helpful for the understanding of specific intermolecular interactions. Molecular surfaces can be used to quantitatively represent a variety of spatially dependent properties. Two types of molecular surface exist: van der Waals surfaces and solvent accessible surfaces. The van der Waals surface [4] is the surface of a sphere determined by the van der Waals radius. The solvent accessible surface of a molecule [19, 20] is the surface that a theoretical water probe touches as it rolls around the van der Waals surface. Surfaces can be drawn using a grid of dots, a wire-mesh, or in solid form. Properties such as electrostatic potential, temperature, or user-defined values can be mapped onto surface drawings.

6.3 Molecular Query Method

For efficient visualization of structures, it is important to provide the ability to manipulate structures flexibly and efficiently. In general, the representations of fully detailed structures are not adequate to illustrate and analyze large and complex structures.

The combinations of different types of representations based on the aspect of the structure of interest and the ability to fine-tune the level of detail of specific parts of a molecule are extremely important in understanding complex structures. Isolation of the area of interest is key to researchers dealing with large biochemical systems. The user needs to select some portion of components with a clear view of which components are selected, and then selectively label, color-code and display at the different abstraction levels. The user also needs to specify the amount of detail to be displayed.

The molecular modeling system requires the ability to query on the structure to support an efficient and flexible subsetting mechanism. A number of basic and complex subsets are required to analyze the features and structure of biological macromolecules. A number of primitive queries can be combined into arbitrarily complex queries connected with a logical operator such as AND, OR, NOT.

Most systems provide selection functions in the form of a command language syntax. The query specifications are limited and can not be extended in these systems. Some researchers have used SQL or developed SQL-like query languages specifically designed for molecular modeling to support complex queries [68]. Though these approaches provide extensive and flexible methods, they are difficult to use and implement. We have developed a different approach using OTMM that is easy to use as well as flexible and extensible.

MMFC provides the class MolSet whose objects hold pointers to a group of objects of any type. The Selector class provides the ability to query on these collection types. Query specifications are based on various types of patterns as described in the previous chapter. Selector is a template class parameterized by Filter and Predicate. Filter is another template class parameterized by Iterator and Predicate. Filter provides the ability to

iterate only over those elements that satisfy a particular condition determined by

Predicate, which is a function object.

```
template <class MCIter, class Predicate>
class Selector {
public:
    Selector(const Filter<MCIter, Predicate>& filt) : _filter(filt);
    MolSet Select();

private:
    Filter<MCIter, Predicate> _filter;
};
```

The Select member function uses Filter to perform a selection and returns the result in a MolSet class.

```
MolSet Select() {
    MolSet ms;
    while ( _filter) {
        ms.Add(**_filter);
        ++_filter;
    }
    return ms;
}
```

An example of Predicate that determines whether an atom is within a specific distance from the reference atom is:

```
class IsWithin {
public:
    IsWithin(Atom* ref, float dist)
        : _reference(ref), _distance(dist) {}
    bool operator()(const Atom& atom) const;
private:
    Atom* _reference;
    float _distance;
};

bool IsWithin::operator()(const Atom& atom) const {
    Vector v1 = atom.Point();
    Vector v2 = _reference->Point();
    return Length(v1-v2) < _distance;
}
```

The user will apply Selector to select all the atoms that are within 5 Å of a reference atom.

```
Compound compd;
Atom* ref;
Filter f(AtomIterator(compd), IsWithin(ref, 5));
Selector sel(f);
MolSet ms = sel.Select();
```

Now we want to select only the CA atoms that are within 5 Å of a reference atom. We

define a Predicate that determines whether an atom has a specific name:

```
class IsName {
public:
    IsName(char* name) : _name(name) {}
    bool operator()(const Atom& atom) const;
private:
    char* _name;
};

bool IsName::operator()(const Atom& atom) const {
    return !strcmp(_name, atom.Name());
}
```

Since the MolSet class supports standard logical operations such as AND, OR, NOT, which are implemented by the operators &, +, and ~, respectively, selections with complex queries combined with logical operators are supported without additional coding.

```
Compound cmpd;
Atom* ref;
Filter f1(AtomIterator(cmpd), IsWithin(ref, 5);
Filter f2(AtomIterator(cmpd), IsName("CA");
Selector sel1(f1);
Selector sel2(f2);
MolSet ms = sel1.Select() & sel2.Select();
```

Using Selector in conjunction with Filter and various Predicates, we can design an efficient molecular query methodology with unlimited flexibility and extensibility. If one needs a query based on a new pattern, all one has to do is to write a Predicate that implements the new pattern. A new pattern implemented in the form of a module can be easily added to the existing system. Furthermore, users do not have to learn an extra query language with this approach.

6.4 Applications

In this section, we design application modules for various structural visualizations discussed above. Visualization is an important application area with an extensive set of algorithms. We concentrate on how OTMM can be used in general for the common

visualization requirements in molecular modeling, and how new algorithms can be efficiently plugged into the environment.

There are two key operations in the visualization process of molecular modeling. First, the molecular components are translated to geometric components such as points, lines, splines, spheres, polygons, tubes, texts, and surfaces. Various transformer modules including Line, Sphere, and Ribbon are developed for this purpose. Once the geometric primitives have been defined, they are rendered with the appropriate rendering parameters including position, coloring, illumination, and surface properties. 3D Viewer embedded in Pro3D is responsible for this operation.

Some important display attributes associated with the visualization are rendering style, coloring and labeling method. A rendering style determines the primitives used to draw the atoms, bonds, and other molecular components. Different rendering styles are available for different components as listed below:

- Atoms: dots, sphere
- Bonds: line, cylinder
- Ribbons: multiple parallel lines, 2-D flat surface, 3-D solid model
- Secondary structures: plain, schematic
- Surface: dots, triangle strips

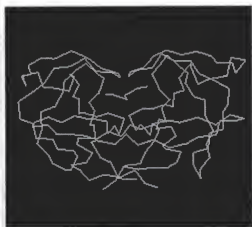
Color coding can be used to distinguish or highlight particular atoms or residue types in a structure. The surfaces can be colored according to atom, surface component, or various properties such as occupancy, mass, and charges. The use of assigned color as a property can act as a bridge between atomic and surface properties. For example, suppose we want to find all vertices of a surface which are concave, that is, having negative surface curvature, and which are formed by hydrophobic residues. One way to do this is to color all hydrophobic residues one color, then transfer that color to the surface, then select all vertices which have that color AND which have curvature less than zero.

Another important property related to the coloring method is the transparency. Inclusion of transparency effects allows one to view the inside of a large molecule. One can display both the surface and structure simultaneously by adjusting the degree of the surface transparency.

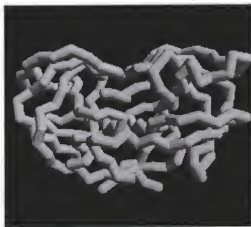
Labels are useful for various purposes. They help a viewer identify the specific components of a structure drawing such as atoms or residues. They are also used to view data associated with each atom such as atom type and partial atomic charge, and to view geometric measurements associated with particular sets of atoms such as bond lengths and bond angles.

We have designed several application modules for the visualization including PDBLoad, Select, GCGenerator, and 3-D Viewer. The 3-D structure of a molecule is known in atomic detail through data available from a variety of sources. The primary source for 3-D molecular structure data is the Brookhaven Protein Data Bank (PDB), which is distributed in the form of ASCII files with one structure per file. PDBLoad is a Producer module that reads PDB files and creates molecular components to be visualized.

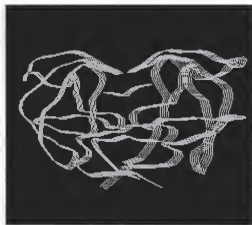
GCGenerator is a collection of modules that convert molecular components into geometry components in various types including line, tube, ball-and-stick, space filling, and ribbon. Many flexible representations are possible by selectively combining different models using Select Module and Operator Module. These geometry components are rendered by the 3-D Viewer Module that also provides abilities to rotate, translate and scale the object in real time. An HIV protease molecule displayed with a number of different models is shown in Figure 6.3. An example of a module network to produce a combination of various display styles within a single molecule is given in Figure 6.4 and the result is illustrated in Figure 6.5.



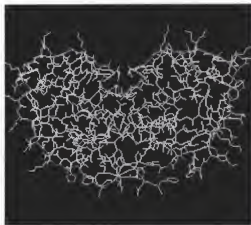
(a) Backbone line model



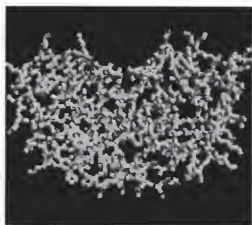
(b) Tube model



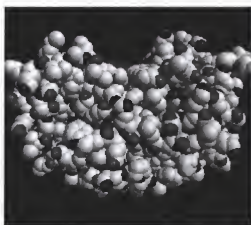
(c) Ribbon model



(d) Line model



(e) Ball and stick model



(f) Space filling model

Figure 6.3. HIV protease in different display models

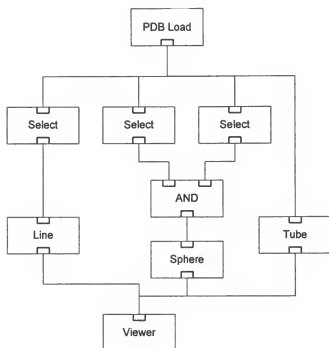


Figure 6.4. A module network

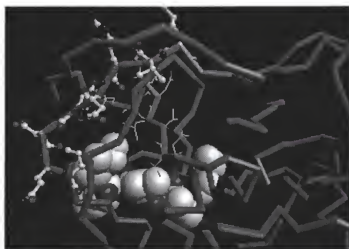


Figure 6.5. Output of the module network in Figure 6.4

The above example shows how the visual programming method equipped with various modules helps the user create the combinations of different types of representations with the ability to fine-tune the level of detail of specific parts of a

molecule, which are extremely important in understanding complex structures of biomolecules. The user can select some portion of components, and then selectively label, color-code and display the different styles at will.

CHAPTER 7 MOLECULAR SIMULATIONS

7.1 Introduction

Molecular simulation methods provide powerful computational tools to predict the structure and movements of molecules by calculating molecular energies and properties associated with structural conformation. The three primary classes of molecular simulation methods are molecular mechanics [9, 10], molecular dynamics [32, 33, 52, 75], and normal mode analysis [45].

Molecular mechanics is a computational method designed to give *a priori* structures and energies for molecules. The greatest advantage of molecular mechanics is its computational speed, which allows for its use in procedures such as energy minimization and conformational energy searching that require numerous energy evaluations. In this method, the molecule is viewed as a collection of points (atoms) connected by springs (bonds) with different elasticities (force constants). Each of the individual energy terms have preferential equilibrium positions and force constants that are either experimentally known or theoretically estimated and used to associate energetic penalties with each individual deviation.

Molecular dynamics simulates the motion of the system and produces a collection of positions and velocities which describe the movement of the system. Atoms in biomolecules are in a state of constant motion at room temperature. The x-ray structure of a protein only provides a time-averaged structure of a continuously moving system. At any given time, a typical protein exhibits a wide variety of motions. Molecular dynamics

simulations, providing the actual motion of the system, can be used to analyze local fluctuations of atoms and to study the path and time dependence of transitions between conformational states.

In normal mode analysis, the motion is described as a superposition of harmonic vibrations whose characteristics are determined by the shape of the potential surface near an energy minimum. This approach has the advantage that, once the modes of vibration are determined, many time average and dynamic properties can be computed easily. However, calculations of the matrix eigenvalues for full determination of the normal mode require tremendous computing time. Furthermore, this method does not allow a detailed treatment of solvation phenomena.

In this chapter, we discuss the theoretical backgrounds of two simulation methods: molecular mechanics and molecular dynamics, and design the application component using OTMM. The essential part of molecular simulation is to compute the energy of a molecule, and find the structure corresponding to the minimum energy. The process of molecular simulation involves the transformation of basic physical equations into function objects.

7.2 Force Fields

To apply the simulation methods to a macromolecular system, one must have a knowledge of the potential energy surface and its gradients. The combination of potential energy functions and their associated sets of numerical parameters is called the force-field. Many different kinds of force-fields have been developed by different research groups, two of the more popular programs for macromolecules being AMBER [60] and CHARMM [8]. We use the force-field from CHARMM in this application, however any force-field can be used in the same way. Furthermore, multiple force-fields can be used simultaneously.

Potential energy of the system is expressed as a function of the atomic coordinates. The forces acting on the atoms of the system are obtained from the first derivatives of the potential with respect to the atom positions. These forces can be used to calculate dynamic properties of the system. From the second derivatives of the potential surface, the force constants for small displacements can be evaluated and used to find the normal modes.

7.2.1 Potential energy

In the CHARMM force-field, the total energy is composed of internal energy terms, nonbonded interactions, and constraints energy, which is given by

$$E = E_b + E_\theta + E_\phi + E_\omega + E_{vdw} + E_{el} + E_{hb} + E_{c_r} + E_{c_\phi} \quad (7.1)$$

The definition of each term is discussed below.

(1) Internal energy terms

Internal energy arises from deviations from ideal structural features. Force constants (k_x) and geometric constants (r_0 , θ_0 , ω_0) are selected from the parameter table based on the atom types that are involved.

- Bond potential: the energy of a bond stretched or compressed from its natural length

$$E_b = \sum_{bonds} k_b (r - r_0)^2$$

- Bond angle potential: the energy of bending a bond angle from its natural value

$$E_\theta = \sum_{angles} k_\theta (\theta - \theta_0)^2$$

- Torsion potential: the energy due to twisting about a bond

$$E_\phi = \sum_{dihedrals} k_\phi (1 - \cos n\phi) \quad \text{where } n = 1-6$$

- Improper torsion potential: the energy of bending planar atoms out of plane

$$E_{\omega} = \sum k_{\omega} (\omega - \omega_0)^2$$

(2) *Non-bonded interactions*

The non-bonded energy represents the pair-wise sum of the energies of all possible interacting non-bonded atoms. The non-bonded energy accounts for van der Waals attraction and electrostatic interactions. The general shape of the van der Waals potential function results from two forces: a repulsive one at short distance, and an attractive one at a greater distance, which rapidly dies off as the interacting atoms move apart. Repulsion occurs when the distance between interacting atoms becomes less than the sum of their contact radii. The curve is characterized mainly by the minimum energy distance which is referred to as the van der Waals radii, the depth of the potential well (ϵ) which is related to the polarizabilities, and the steepness of the repulsive part.

The electrostatic interaction is modeled using a Coulombic potential. The electrostatic energy is a function of the charge on the non-bonded atoms, their interatomic distance, and a molecular dielectric expression that accounts for the attenuation of electrostatic interaction by the environment. For a typical protein, the non-bonded interactions account for most of the energy evaluation time.

- Van der Waals energy: the energy due to van der Waals non-bonded interactions

$$E_{vdw} = \sum_{nonbonds} \left(\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^6} \right) \quad (7.2)$$

- Electrostatic energy: the energy due to electrostatic interactions

$$E_{el} = \sum_{nonbonds} \frac{q_i q_j}{4\pi\epsilon_0 r_{ij}} \quad (7.3)$$

(3) *Constraints*: the energy associated with distance and angle constraints

For some purposes, it is useful to restrict the changes that occur in a structure. Energy minimizations with empirical energy functions may result in conformations that deviate

significantly from the correct one. Constraining the structure spatially yields converged structures while providing control over the deviation from the starting structure. In CHARMM, constraints are implemented by adding constraint energy terms as penalty functions.

Two constraint energy terms are used in CHARMM: atom harmonic constraint and dihedral constraint. Atom harmonic constraints are used to avoid large displacement of atoms when minimizing, while still allowing the structure to relax. Dihedral constraints are used to maintain certain local conformation or when a series of different conformations needs to be examined.

Atom harmonic constraints:

$$E_{c_r} = \sum K_i (r_i - r_{i0})^2 \quad (7.4)$$

Dihedral constraints:

$$E_{c_\phi} = \sum K_i (\phi_i - \phi_{i0})^2 \quad (7.5)$$

The total number of terms in the potential energy function is dominated by the number of non-bonded interactions, which is the complexity of $O(n^2)$ where n is the number of atoms. The interactions are usually turned off at a cutoff distance to reduce the number of interacting pairs and hence the time of computation. Two basic approaches have been used to avoid the discontinuous truncation of the potential at the cutoff distance: smooth switching of the potential over a transition interval and continuous shifting of the potential at all distances such that its value and derivative are zero at the cutoff. A switching function used in CHARMM for the first approach is defined by

$$sw(r, r_{on}, r_{off}) = \begin{cases} 1 & r \leq r_{on} \\ \frac{(r_{off}^2 - r^2)^2 (r_{off}^2 + 2r^2 - 3r_{on}^2)}{(r_{off}^2 - r_{on}^2)^3} & r_{on} < r \leq r_{off} \end{cases}$$

This function satisfies $sw(r_{on}) = 1$, $sw(r_{off}) = 0$, $dsw/dr(r_{on}) = 0$, and $dsw/dr(r_{off}) = 0$, yielding a continuous potential energy and force. In the second approach a number of different functions are used. A shifted potential modifies the function so that the energy and forces go to zero at cutoff distance using the function

$$S(r) = 1 - 2r^2/r_{off}^2 + r^4/r_{off}^4, \quad r \leq r_{off}$$

One drawback to using this function is that it has a discontinuity in the second derivative at the cutoff distance. A force shifted potential offsets the atom-atom force from the true Coulomb force by a constant using the function

$$S(r) = (1 - r/r_{off})^2, \quad r \leq r_{off}$$

7.2.2 Gradients

The evaluation of gradients is required for energy minimizations and molecular dynamics simulations. Force is the first derivative of the potential energy function with respect to the Cartesian coordinates of the functions involved in a specific interaction term. Certain minimization techniques, such as Newton's method, require the second derivatives as well. The second derivative of the energy is obtained from the Hessian matrix which is constructed by finite differences from the displacement and forces.

The derivatives of the bond and angle potential are found by a straightforward method. The force due to a bond potential formed by atoms (i , j) is represented by

$$\mathbf{F}_i = -\hat{\mathbf{r}}_j 2k_b(r_{ij} - r_0), \quad \mathbf{F}_j = -\mathbf{F}_i$$

where $r = \|\mathbf{r}\|$, and $\hat{\mathbf{r}} = \frac{\mathbf{r}}{\|\mathbf{r}\|}$.

The force due to an angle potential formed by three atoms (i , j , k) is represented by

$$\mathbf{F}_i = 2k_\theta(\theta - \theta_0)\mathbf{A}, \quad \mathbf{F}_j = -(\mathbf{F}_i + \mathbf{F}_k), \quad \mathbf{F}_k = 2k_\theta(\theta - \theta_0)\mathbf{B}$$

where

$$\mathbf{A} = -\frac{1}{r_{ji} \sin \theta} (\hat{\mathbf{r}}_{jk} - \hat{\mathbf{r}}_{ji} \cos \theta) \quad \mathbf{B} = -\frac{1}{r_{jk} \sin \theta} (\hat{\mathbf{r}}_{ji} - \hat{\mathbf{r}}_{jk} \cos \theta)$$

The torsion energy is not differentiated with respect to ϕ because of singularities when angles become planar. The derivatives of the torsional potential are found by multiple use of the chain rule, and it is differentiated with respect to $\cos \phi$. For the same reason, the derivatives of planar improper torsions are expanded in a Taylor series when ω is small. When ω is large, a straightforward approach is used. The functional form of the first derivative of torsional potential formed by four atoms (i, j, k, l) is given by

$$\mathbf{F}_i = -k_\phi \frac{\partial \mathcal{V}}{\partial \cos \phi} \mathbf{R} \times \mathbf{r}_{kj}$$

$$\mathbf{F}_j = k_\phi \frac{\partial \mathcal{V}}{\partial \cos \phi} (\mathbf{R} \times \mathbf{r}_{ji} + \mathbf{S} \times \mathbf{r}_{kl}) - \mathbf{F}_l$$

$$\mathbf{F}_k = -k_\phi \frac{\partial \mathcal{V}}{\partial \cos \phi} (\mathbf{R} \times \mathbf{r}_{ji} + \mathbf{S} \times \mathbf{r}_{kl}) - \mathbf{F}_l$$

$$\mathbf{F}_l = -k_\phi \frac{\partial \mathcal{V}}{\partial \cos \phi} \mathbf{S} \times \mathbf{r}_{kj}$$

where

$$\frac{\partial \mathcal{V}}{\partial \cos \phi} = \begin{cases} 1 & n = 1 \\ 4 \cos \phi & n = 2 \\ 12 \cos^2 \phi - 3 & n = 3 \\ 32 \cos^3 \phi - 16 \cos \phi & n = 4 \\ 80 \cos^4 \phi - 60 \cos^2 \phi + 5 & n = 5 \\ 192 \cos^5 \phi - 192 \cos^3 \phi + 36 \cos \phi & n = 6 \end{cases}$$

$$\mathbf{R} = \frac{1}{m} (\hat{\mathbf{n}} - \hat{\mathbf{m}} \cos \phi), \quad \mathbf{S} = \frac{1}{n} (\hat{\mathbf{m}} - \hat{\mathbf{n}} \cos \phi)$$

$$\mathbf{m} = \mathbf{r}_{ji} \times \mathbf{r}_{kl}, \quad \mathbf{n} = \mathbf{r}_{kl} \times \mathbf{r}_{kj}$$

The force due to a van derWaals term by atom pair (i, j) without using cutoff function is given by

$$\mathbf{F}_i = -\hat{\mathbf{r}}_{ij} \frac{dE}{dr} = -\hat{\mathbf{r}}_{ij} \frac{12\varepsilon}{r} \left(\frac{\sigma^3}{r^6} - \frac{\sigma^6}{r^{12}} \right), \quad \mathbf{F}_j = -\mathbf{F}_i$$

7.2.3 Force-Field Function Objects

The various force-fields generally contain not only different parameters, but also different potential functions, which make it difficult to transfer parameters from one force field to another. While it is easy to incorporate changed parameters into a calculation, changing a functional form is a sizable reprogramming task. Consequently, most available programs are tailored to one force field, or contain a number of different potential functions from which a user can make a choice. However, the set of available force fields is still fixed in these programs and users cannot try customized versions. Programming with OTMM addresses this problem by enabling users to add customized force fields with a minimum effort.

Each potential energy and gradient equation is implemented as a function object. The following sample code illustrates a function object that calculates bond energy.

```
class BondEnergy {
public:
    double operator()(const Bond& bond) const;
private:
    ParamMgr _pMgr;
};

double BondEnergy::operator()(const Bond& bond) const {
    float F = _pMgr->Force(bond);
    float S = _pMgr->Value(bond);
    double s = bond.Length();
    double db = s - S;
    return F * db * db;    // Energy = F * (s - s0)**2
}
```

Here a bond is passed as an argument to `operator()()`, and the energy of a single bond is returned. Force field constants corresponding to the specific bond are returned by `ParamMgr`. Although some equations look complicated in their forms, the

implementation of an individual equation is straightforward since MMFC provides reusable classes with all the basic operations required for the computations including parameter searching, standard vector and matrix operations, and computing values of components such as bond length, angle and so on.

Once all the function objects for energy and gradient are written, the framework provides the mechanism in which these primitive function objects are deployed for larger functions. The total bond energy and force of the structure is given by

```
double energy = Accumulate(BondIterator(compound), BondEnergy);
```

The design principle of MMFC is well illustrated in this single line code. The three different types of objects: a template function (`Accumulate`), an Iterator (`BondIterator`), and a function object (`BondEnergy`), are appropriately combined to compose a larger function and operated on a structure. All that the application programmer has to write is a primitive function object, `BondEnergy`, which is a straightforward implementation of a specific equation.

The advantage of using function objects becomes more evident when it comes to the calculation of nonbonded interactions. In the calculation of nonbonded interactions, a cutoff function such as a switch or shift may be used to turn off interactions at some cutoff distance. The characteristic of a cutoff function is that the functional form becomes different depending on the distance of two atoms. Furthermore, a number of different cutoff functions can be selected. Incorporating the several cutoff functions into the `Energy` or `Force` function makes the code cumbersome and cutoff function dependent. The following pseudo code shows what the `Energy` function looks like with a traditional approach.

```
// Select cutoff function based on option
switch (option)
case A:
    cutofff1 = fa(r, ron, roff) // r < ron
    cutofff2 = ga(r, ron, roff) // ron < r < roff
```

```

    cutoff3 = ha(r, ron, roff) // roff < r
case B:
    cutoff1 = fb(r, ron, roff)
    cutoff2 = gb(r, ron, roff)
    cutoff3 = hb(r, ron, roff)
case C:
    .....

// Energy is computed without cutoff function and stored in e
// Now apply cutoff function
if (r < ron)
    e = e * cutoff1
else if (ron < r < roff)
    e = e * cutoff2
else
    e = e * cutoff3

```

With function objects, functions are treated like values, that is the result of a function application can be another function. We define a function object `Cutoff` that returns an appropriate function based on the bond length. The `Energy` function now will simply look like

```

// Assume energy is already computed in e
e = e * Cutoff(r, ron, roff)

```

The `Energy` function contains no cutoff function specific code. `Cutoff` function itself is passed as an argument and returns an actual cutoff function.

7.3 Energy Minimization

The thermodynamically stable states of a molecule correspond to its lowest energy conformations. The energy is a function of the atomic coordinates and the goal of energy minimization is to find the coordinates which correspond to an energy minimum. Energy minimization relaxes the structure by iteratively moving the atoms down the energy gradient. For macromolecules, these methods can explore only a tiny fraction of conformational space, and are generally unable to find the global minimum. Only a local minimum will be found, which is the one closest to the starting point.

There are several different algorithms available [39, 40], all of which use iterative algorithms in which the atomic coordinates are modified from one iteration to the next in

order to decrease the energy based on the gradient information. Each of these approaches has tradeoffs between computational requirements, efficiency and avoidance of mathematical problems. The primary issues are tolerance to poorly behaved regions of an energy function, stability, reliability and precision. Two first-order algorithms, the steepest descent and conjugate gradient methods, are most commonly applied to energy minimization in macromolecular modeling. In practical application, several methods are successively applied to complete a minimization that was initiated by one method. Some of the more commonly used minimization algorithms are discussed below.

7.3.1 Steepest Descent Method

The steepest descent method searches directions based on the local slope of the energy surface. The gradient vector points in the direction of steepest ascent while the negative of the gradient vector points in the direction of steepest descent. In each step of iteration, the coordinates are adjusted in the negative direction of the gradient.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mu \nabla f(\mathbf{x}_k)$$

where $\mathbf{x}$ is a coordinate vector, ∇ is a gradient vector, and μ is a step size.

The step size is increased if a lower energy results; otherwise, it is decreased. The process is continued until some predetermined stopping condition is satisfied. Some common stopping conditions are

- 1) Step tolerance: stop when $\mu < \varepsilon$
- 2) Gradient tolerance: stop when $\|\nabla f(\mathbf{x}_k)\| < \varepsilon$
- 3) Function tolerance: stop when $f(\mathbf{x}_k) - f(\mathbf{x}_{k+1}) < \varepsilon$
- 4) Number of iterations: stop when number of iterations $> N$

This procedure works best when the gradient is large (far from a minimum), but overall convergence properties are poor. It is useful to eliminate large steric conflicts that usually exist in the early stage of minimization.

7.3.2 Conjugate Gradient Method

The conjugate gradient method searches the minimum of a function with a set of conjugate direction vectors. This method begins with the following definition.

Definition Let A be a symmetric matrix of order n , then two vectors v_1 and v_2 are said to be A -orthogonal (A -conjugate) if and only if $v_1^T A v_2 = 0$.

A -orthogonal vectors are a linearly independent set, that is

$$\sum_{i=0}^{n-1} \alpha_i v_i = 0$$

implies that all $\alpha_i = 0$, $i = 0$ to $n-1$.

Consider the problem of minimizing a quadratic function

$$f(x) = \frac{1}{2} x^T A x + b^T x + c$$

where A is a positive definite symmetric matrix.

The solution x^* satisfies the equation

$$\nabla f(x) = A x^* + b = 0$$

Suppose that $\{v_0, v_1, \dots, v_{n-1}\}$ is a set of A -orthogonal vectors. Since $\{v_0, v_1, \dots, v_{n-1}\}$ is linearly independent, x^* can be expressed as

$$x^* = \sum_{i=0}^{n-1} \alpha_i v_i$$

Thus,

$$-b = A x^* = A \sum_{i=0}^{n-1} \alpha_i v_i = \sum_{i=0}^{n-1} \alpha_i A v_i$$

Multiplying v_j^T on both sides yields

$$-v_j^T b = v_j^T \sum_{i=1}^{n-1} \alpha_i A v_i = \sum_{i=1}^{n-1} \alpha_i v_j^T A v_i = \alpha_j v_j^T A v_j$$

Since A is positive definite

$$\alpha_j = \frac{-v_j^T b}{v_j^T A v_j}$$

and

$$x^* = \sum_{i=1}^{n-1} \alpha_i v_i = \sum_{i=1}^{n-1} \frac{-v_i^T b}{v_i^T A v_i} v_i$$

From the above equations, an optimal solution x^* can be found if we know a set of A -

orthogonal vectors $\{v_0, v_1, \dots, v_{n-1}\}$. The key procedure of the conjugate gradient

algorithm is to find vectors $\{v_0, v_1, \dots, v_{n-1}\}$ iteratively. There is an infinite number of

ways to select such vectors. The algorithm implemented in CHARMM is a method

known as the Fletcher-Reeves method. It is summarized in the following steps.

- 1) Compute $v_0 = -\nabla f(x_0)$ for any initial value x_0 .
- 2) Set $x_{k+1} = x_k + \alpha_k v_k$, where α_k minimizes $f(\alpha) = f(x_k + \alpha v_k)$ along the direction $\{x_k + \alpha v_k : \alpha \geq 0\}$ extending from x_k in the direction v_k (a line search is used in this step). Here, $x_k = [x_{k_1}, \dots, x_{k_n}]^T$.
- 3) Compute $f(x_{k+1})$ and $\nabla f(x_{k+1})$.
- 4) If $k+1 = n$, then go to Step 5. Otherwise, set

$$v_{k+1} = -\nabla f(x_{k+1}) + \frac{\nabla f(x_{k+1})^T \nabla f(x_{k+1})}{\nabla f(x_k)^T \nabla f(x_k)} v_k$$

and return to Step 2.

- 5) Set $x_0 = x_n$, and repeat from Step 1 until stopping condition is satisfied.

The conjugate gradient method is more efficient in that it combines information on the current gradient with that based on the gradient at previous steps. This method

converges faster than the steepest descent, however, the user may encounter problems when the initial conformation is far from a minimum.

The Powell method is similar to the conjugate gradient method, but uses more advanced rules to determine the descent direction. As a result, it is much faster than the conjugate gradient method and is well suited for a wide variety of problems.

7.3.3 Adopted-Basis Set Newton-Raphson Method

This technique uses the second derivative Hessian matrix, which is diagonalized and the positive eigenvalues are inverted. Steepest descent steps are applied along the directions of zero or negative eigenvalues. Expanding a function up to the second order Taylor series

$$f(x) = f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2} f''(x_k)(x - x_k)^2$$

To minimize $f(x)$, we set

$$f'(x) = f'(x_k) + f''(x_k)(x - x_k)$$

Solving for x and generalizing in a vector form, we have

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mu \mathbf{R}^{-1} \nabla f(\mathbf{x}_k)$$

where $\mathbf{R}^{-1}$ is an inverse Hessian matrix.

This algorithm may find a minimum in fewer steps than the gradient-only methods, however, it is unsuitable for the highly nonquadratic energy surfaces with multiple local minima and computation requirement is very high.

7.3.4 Minimizer

Minimizer is an abstract class that provides the common control flow for various minimization algorithms. Though different algorithms use different mathematical procedures, each algorithm uses a variation of the exact same loop, changing only the

criteria by which gradients are updated and when to exit. Minimizer defines the skeleton of any minimization algorithms, deferring some algorithm-specific steps to subclasses using Template method pattern. A template method defines an algorithm in terms of invariant parts and abstract operations that subclasses override to implement the behavior that can vary. The following simplified code shows the basic minimization algorithm that the Minimizer class defines:

```
class Minimizer {
public:
    void Run();
    double TotalEnergy() {return _lowestEnergy;}

protected:
    Minimizer(Compound&, int numIter, double stepSize);
    virtual void Update() = 0;
    virtual bool IsExit() = 0;

    Compound& _cmpd;
    double _energy, _prevEnergy, _lowestEnergy;
    double _stepSize;
    int _numIterations;
    int _currentIterations;
};

void Minimizer::Run() {
    while (!IsExit()) { // template method

        // Compute Force-Field
        _energy = Accumulate(BondIterator(_cmpd), BondForce());
        _energy += Accumulate(AngleIterator(_cmpd), AngleForce());
        .....

        Update(); // template method
        ++_currentIterations;
    }
}
```

Two template methods, Update and IsExit, are defined as pure virtual functions in the base class, and overridden by subclasses to implement subclass-specific operations.

The template method enables the base class to define the control structure and to be reused by different subclasses without concern about the control structure. A subclass SDMinimizer is derived from the Minimizer class to implement a steepest descent algorithm.

```
class SDMinimizer : public Minimizer {
```

```

public:
    SDMinimizer(Compound& compd, int numIter, double stepSize)
        : Minimizer(compd, numIter, stepSize) {}

protected:
    virtual void Update();
    virtual bool IsExit();
};

```

An Update defines the main body of the steepest descent algorithm updating coordinates of structure using the gradient information.

```

void SDMinimizer::Update() {
    if (_energy < _prevEnergy)
        _stepSize *= 1.2;
    else
        _stepSize *= 0.5;

    double norm = 0;
    for (AtomIterator ait(_compd); ait; ++ait) {
        Vector v = (*ait)->Gradient();
        norm += v.x*v.x + v.y*v.y + v.z*v.z;
    }

    norm /= (3 * _compd.NumAtoms());
    double s = _stepSize / sqrt(norm);

    for (AtomIterator ai(_compd); ai; ++ai) {
        Vector v = (*ai)->Point();
        v -= (float)s * (*ai)->Gradient();
        (*ai)->Point(v);
    }
}

```

The final structure obtained from the minimization is sensitive to the criteria used for deciding when to stop the iterative procedure. The criteria include a small energy threshold, a geometry change threshold, and a first-derivative threshold. IsExit method defines the exit strategy that determines when to terminate the procedure optimized for different minimization algorithms. The simplest case is to exit when the algorithm has iterated a certain number of times.

```

bool SDMinimizer::IsExit() {
    if (_currentIterations >= _numIterations)
        return TRUE;
    return FALSE;
}

```

The following sample code shows the use of the template method.

```

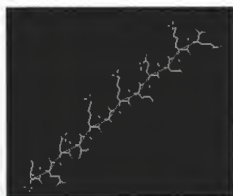
SDMinimizer aMinimizer; // Instance of concrete subclass

```

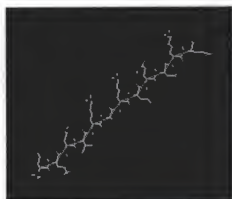
```
aMinimizer.Run(); // Call the base class method
```

We have built two sample structures, and run Minimizer. Both structures have the same 12 amino-acid sequence: ILE GLU ASN VAL LYS ALA LYS ILE GLN ASP LYS GLU, which is taken from Ubiquitine. The first structure is a linear model built with equilibrium values from the CHARMM parameter. The second structure is a helical model built with the coordinates from the PDB data. The structures before and after energy minimization are shown in Figure 7.1 and 7.2. Minimization is performed in the steepest descent method with 1000 iterations. For the linear model, internal energies are almost zero, while nonbond energy is very high. This is because the linear model is an artificial structure model built using equilibrium bond and angle values. For the helical model, internal energies are slightly higher than the linear model, while nonbond energy is very low compared to that of the linear model. Since nonbond energy is a dominating factor, the linear model is much higher in total energy. Total energies have been decreased considerably for both models after minimization, with little noticeable change in structure.

The purpose of this example is to demonstrate how the MMFC can be used to tackle the complex mathematical algorithms in molecular modeling. It requires further investigation to design a full-featured minimization function which can be more effectively applied to describe the structural energetic of actual biological molecules, and which will in fact reflect any changes in conformation which occur.

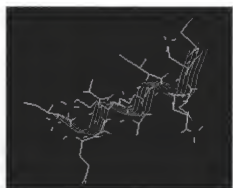


(a) Before energy minimization



(b) After energy minimization

Figure 7.1 Linear model



(a) Before energy minimization



(b) After energy minimization

Figure 7.2 Helical model

CHAPTER 8

CONCLUSION

The primary goal of this research is to develop a software tool which provides developers, scientists, and end users with an ideal environment to develop and use a fully integrated and expandable molecular modeling software system with modularity and interconnectivity. We have developed two key components for this purpose: MMFC and Pro3D. MMFC is a domain-specific, object-oriented framework for developing a molecular modeling system, which provides a set of reusable object-oriented building blocks for molecular modeling software from which new algorithms could be rapidly assembled. Developers could fully take advantage of existing components without reimplementing all of the software infrastructure in each new molecular modeling project. The framework is implemented in the form of a C++ class library which encapsulates a large portion of API required in the broad range of molecular modeling applications. New features can be easily added by deriving new, specialized classes without modifying the existing framework directly.

Pro3D is a full-featured molecular modeling program which enables researchers to build a model of the molecule of interest, apply molecular mechanics and molecular dynamics simulation, and provide powerful graphics and computational tools to efficiently visualize and analyze the structural and dynamic properties of the molecule. Two key features that distinguish Pro3D from the traditional programs are the visual programming and component based environments. To provide developers with a flexible environment for application development, we have introduced the application module

with component software technology. This feature benefits not only developers but also users by providing both compile-time and run-time reuse, and a user interface with a visual programming style. On the developer's side, new functions can be easily implemented in the form of components with the MMFC. The components will all inter-operate with the same look and same feel. Combined with ease of access to the Internet which allows the automatic distribution of such components, this advancement permits collaboration between scientists in different locations with different areas of expertise.

Some of the features unique to this research are summarized as follows:

- Developing a full-featured object-oriented framework for molecular modeling
- Component software design for flexible and expandable application development which allows the user to take advantage of the Internet environment
- Visual programming user interface

There are many potential directions for future research and development. One immediate interest is to expand OTMM by developing more applications modules such as full-featured energy minimization and molecular dynamics simulation. A number of existing algorithms as well as new ones can be easily implemented by taking advantage of reusable components of MMFC. Once developed in the form of a module using MMFC, these algorithms can be seamlessly plugged into the Pro3D environment with uniform interface.

Another possibility is to develop Web-enabled components to take advantage of Web resources from within the application program. The World Wide Web enables researchers working on virtually any computer platform to access remote databases. Web-enabled components which integrate the HTML viewer into the application enable the user to access the Internet without leaving the application, allowing the resources on the Internet

to be fed into other modules automatically, thus forming hybrid applications that combine data and programs from the Internet with local data and programs.

Such a component-based approach allows the developer to build distributed applications across the Internet. Today, two popular technologies that support distributed objects are CORBA (Common Object Request Broker Architecture) from OMG (Object Management Group) and DCOM (Distributed Component Object Model) from Microsoft. Components with distributed object support allow developers to split an application into component modules that can each transparently execute on a different computer connected to the Internet. Any function calls to the remote component would be forwarded automatically to the remote service for execution, and the results would be returned to the calling component.

In OTMM, partitioning an application is no longer an issue, since a network of modules corresponds to an application and each module corresponds to a component. Furthermore, because the module network explicitly represents which modules can be run in parallel, parallelism at a module level can be easily accomplished with distributed objects.

REFERENCES

- [1] ANSI (American National Standards Institute), *American National Standard for Information Processing Systems—Programmer's Hierarchical Interactive Graphics System*, ANSI, X3.144-1988, ANSI, New York, 1988.
- [2] Apple Computer, Inc., Cupertino, CA., *Macintosh Programmers Workshop Pascal 3.0 Reference*, 1989.
- [3] J. J. Barton and L. R. Nackman, *Scientific and Engineering C++*, Addison-Wesley, 1994.
- [4] P. A. Bash, N. Pattabiraman, C. Huang, T. E. Ferrin, and R. Langridge, "Van der Waals Surfaces in Molecular Modeling: Implementation with Real-Time Computer Graphics," *Science*, Vol. 222, pp.1325-1327, Dec, 1983.
- [5] J. A. Board Jr, L. V. Kale, K. Schulten, R. D. Skeel, and T. Schlick, "Modeling Biomolecules: Large Scales, Longer Durations," *IEEE Computational Science & Engineering*, pp.19-30, Winter, 1994.
- [6] G. Booch, *Object-Oriented Analysis and Design with Applications*, The Benjamin/Cummings Publishing Co., 1994.
- [7] J. E. Bowie, *Data Visualization in Molecular Science*, Addison-Wesley, 1995.
- [8] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus, "CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations," *Journal of Computational Chemistry*, Vol.4, No.2, pp.187-217, 1983.
- [9] R. E. Bruccoleri and M. Karplus, "Spatially Constrained Minimization of Macromolecules," *Journal of Computational Chemistry*, Vol.7, No.2, pp.165-175, 1986.
- [10] U. Burkert and N. L. Allinger, *Molecular Mechanics*, ACS Monograph 177, 1982.
- [11] M. D. Carroll and M. A. Ellis, *Designing and Coding Reusable C++*, Addison-Wesley, 1995.
- [12] M. Carson, "RIBBONS 2.0," *Journal of Applied Crystallography*, 24, pp.958-961, 1991.
- [13] D. A. Case, "Computer Simulations of Protein Dynamics and Thermodynamics," *IEEE Computer CS&E*, pp.47-57, Oct, 1993.

- [14] O. Casher, S. M. Green, and H. S. Rzepa, "EyeChem 1.0: A molecular chemistry toolkit for collaborative molecular visualization," *Journal of Molecular Graphics*, Vol.12, pp.226-230, Sep, 1994.
- [15] *Catalogue of CPK Precision Molecular Models*, Ealing Corporation, 22 Pleasant St., S. Natick, Mass., 1981.
- [16] W.Chang, I. N. Shindyalov, C. Pu, and P. E. Bourne, "Design and Application of PDBlib, a C++ Macromolecular Class Library," *CABIOS*, Vol.10, No.6, pp.575-586, 1994.
- [17] P. Chun and W. Jou, "Molecular Conformation of Ubiquitinated Structures and the Implications for Regulatory Function," *Journal of Molecular Graphics*, Vol.10, pp.7-11, Mar, 1992.
- [18] N. C. Cohen, J. M. Blaney, C. Humblet, P. Gund, and D. C. Barry, "Molecular Modeling Software and Methods for Medicinal Chemistry," *Journal of Medicinal Chemistry*, Vol.33, No.3, pp.883-894, Mar, 1990.
- [19] M. L. Connolly, "Solvent-Accessible Surfaces of Proteins and Nucleic Acids," *Science*, Vol.221, No.4612, pp.709-713, Aug, 1983.
- [20] M. L. Connolly, "Analytical Molecular Surface Calculation," *Journal of Applied Crystallography*, 16, pp.548-558, 1983.
- [21] J. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1991.
- [22] J. Coplien and D. Schmidt, *Pattern Languages of Program Design*, Addison-Wesley, 1995.
- [23] A. Dalke, A. Gursoy, W. Humphrey, L. Kale, and M. Nelson, *namd Programmers Guide*, Beckman Institute at the University of Illinois, 1995.
- [24] B. Duncan, T. J. Macke, and A. J. Olson, "Biomolecular Visualization using AVS," *Journal of Molecular Graphics*, Vol.13, pp.271-282, Oct, 1995.
- [25] G. Edwards, "The Design of a Second Generation Visualization Environment," *Proceedings of Computer Graphics 1991*, Blenheim Online, pp.261, 1991.
- [26] T. Eggenschwiler and E. Gamma, "ET++ SwapManager: Using Object Technology in the Financial Engineering Domain," *Object Oriented Programming Systems, Languages, and Applications*, pp.166-177, 1992.
- [27] T. E. Ferrin, C. Huang, L. E. Jarvis, and R. Langridge, "The MIDAS Database System," *Journal of Molecular Graphics*, Vol.6, pp.2-12, Mar, 1988.
- [28] T. E. Ferrin, C. Huang, L. E. Jarvis, and R. Langridge, "The MIDAS Display System," *Journal of Molecular Graphics*, Vol.6, pp.13-27, Mar, 1988.
- [29] K. Flurchick and L. Bartolotti, "Visualizing Properties of Atomic and Molecular Systems," *Journal of Molecular Graphics*, Vol.13, pp.10-13, Feb, 1995.

- [30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [31] S. Gibbs, "Composite Multimedia and Active Objects," *Object Oriented Programming Systems, Languages, and Applications*, pp.97-112, 1991.
- [32] J. M. Goodfellow, *Molecular Dynamics: Applications in Molecular Biology*, 1990.
- [33] W. F. van Gunsteren and H. J. C. Berendsen, "Algorithms for Macromolecular Dynamics and Constraint Dynamics," *Molecular Physics*, Vol.34, pp.1311-1327, 1977.
- [34] F. Hopgood, D. Duce, J. Gallop, and D. Sutcliffe, *Introduction to the Graphical Kernel System (GKS)*, Academic Press, London, 1986.
- [35] R. E. Hubbard, R. Flettnick, *Computer Graphics and Molecular Modeling*, Cold Spring Harbor, 1986.
- [36] W. Humphrey, A. Dalke, and K. Schulten, *VMD Programmer's Guide*, Beckman Institute at the University of Illinois, 1995.
- [37] W. Humphrey, A. Dalke, and K. Schulten, "VMD: Visual Molecular Dynamics," *Journal of Molecular Graphics*, Vol.14, pp.33-38, Feb, 1996.
- [38] H. Huni, R. Johnson, and R. Engel, "A Framework for Network Protocol Software," *Object Oriented Programming Systems, Languages, and Applications*, pp.358-369, 1995.
- [39] M. E. Jerrell, "Function Minimization and Automatic Differentiation Using C++," *Object Oriented Programming Systems, Languages, and Applications*, pp.169-173, 1989.
- [40] M. W. Jeter, *Mathematical Programming: An Introduction to Optimization*, Pure and Applied Mathematics, 1986.
- [41] T. A. Jones, M. Kjeldgaard, *O V5.1 Manual*, Uppsala University, Sweden, 1995.
- [42] T. A. Jones, "A Graphics Model-Building and Refinement System for Macromolecules," *Journal of Applied Crystallography*, 11, pp.268-272, 1978.
- [43] W. Jou, "On Control of Motion Path and Speed in a Spline-Based Animation," *Ph.D. Dissertation*, University of Florida, 1991.
- [44] W. Jou and P. Chun, "Molecular Mechanics of the Formation of Cholic Acid Micelles," *Journal of Molecular Graphics*, Vol.9, pp.237-240, Dec, 1991.
- [45] P. Kollman, "Free Energy Calculations: Applications to Chemical and Biochemical Phenomena," *Chemical Review*, Vol.93, No.7, pp.2395-2417, 1993.
- [46] G. E. Krasner and S. T. Pope, "A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, pp.26-49, Aug/Sep, 1988.

- [47] P. J. Kraulis, "MOLSCRIPT: a Program to Produce Both Detailed and Schematic Plots of Protein Structures," *Journal of Applied Crystallography*, 24, pp.946-950, 1991.
- [48] A. M. Lesk, *Protein Architecture*, IRL Press, 1991.
- [49] A. M. Lest and K. D. Hardman, "Computer-Generated Schematic Diagrams of Protein Structures," *Science*, Vol.216, No.30, pp.539-540, Aug, 1982.
- [50] M. Linton, "Composing User Interfaces with Interviews," *IEEE Computer*, pp.8-22, Feb, 1989.
- [51] B. Lucas, G. D. Abram, N. S. Collins, D. A. Epstein, D. L. Gresh, and K. P. McAuliffe, "An Architecture for a Scientific Visualization System," *Proceedings of Visualization '92*, IEEE Computer Society Press, pp.107-114, 1992.
- [52] J. A. McCammon, S.C.Harvey, *Dynamics of Proteins and Nucleic Acids*, Cambridge University Press, 1987.
- [53] S. Meyers, *More Effective C++*, Addison-Wesley, 1996.
- [54] R. B. Murray, *C++ Strategies and Tactics*, Addison-Wesley, 1993.
- [55] D. R. Musser and A. Saini, *STL Tutorial and Reference Guide*, Addison-Wesley, 1996.
- [56] D. Musser and A. Stepanov, "Algorithm-oriented Generic Libraries," *Software Practice and Experience*, Vol.24(7), pp.623-642, Jul, 1994.
- [57] Neider, Jackie, Tom Davis, and Mason Woo, *OpenGL Programming Guide*, Addison-Wesley, 1993.
- [58] A. Nicholls, K. Sharp, and B. Honig, "Protein Folding and Association: Insights from the Interfacial and Thermodynamic Properties of Hydrocarbons," *Proteins: Structure and Functions*, 11, pp.282-290, 1991.
- [59] A. J. Olson, D. S. Goodsell, "Visualizing Biological Molecules," *Scientific American*, pp.76-81, Nov, 1992.
- [60] D. Pearlman, D. A. Case, J. W. Caldwell, W. S. Ross, T. E. Cheatham, D. M. Ferguson, G. L. Seibel, U. C. Singh, P. Weiner, and P. A. Kollman, *Amber V4.1 Manual*, University of California at San Francisco, 1995.
- [61] F. M. Richards, "The Protein Folding Problem," *Scientific American*, Jan, pp.54-63, 1991.
- [62] J. S. Richardson, "The Anatomy and Taxonomy of Protein Structure," *Advances in Protein Chemistry* 34 (2), pp.167-339, 1981.
- [63] J. Rumbaugh, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
- [64] R. Sayle, *RasMol: Molecular Visualisation Program*, Glaxo Research and Development, 1992.

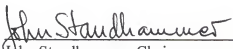
- [65] G. E. Schulz and R. H. Schirmer, *Principles of Protein Structure*, Springer-Verlag, 1979.
- [66] *ScriptX Technical Overview*, Kaleida Labs, 1995.
- [67] J. Shapiro, *A C++ Toolkit*, Prentice Hall, 1991.
- [68] I. N. Shindyalov, W. Chang, C. Pu, and P. E. Bourne, "Macromolecular Query Language (MMQL): Prototype Data Model and Implementation," *Protein Engineering*, Vol.7, No.11, pp.1311-1322, 1994.
- [69] P. S. Strauss, "IRIS Inventor, A 3D Graphics Toolkit," *Object Oriented Programming Systems, Languages, and Applications*, pp.192-200, 1993.
- [70] B. Stroustrup and M. A. Ellis, *The Annotated C++ Reference Manual*, Addison Wesley, 1990.
- [71] M. Teschner, C. Henn, H. Vollhardt, S. Reiling, and J. Brickmann, "Texture Mapping: A New Tool for Molecular Graphics," *Journal of Molecular Graphics*, Vol.12, pp.98-105, Jun, 1994.
- [72] C. Upson, T. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam, "The Application Visualization System: A Computational Environment for Scientific Visualization," *IEEE CG & A*, pp.30-42, 1989.
- [73] W. van Gunsteren and H. Berendsen, *Groningen Molecular Simulation (GROMOS) Manual*, Biomos b.v., 1988.
- [74] J. M. Vlissides and M. A. Linton, "Unidraw: A Framework for Building Domain-Specific Graphical Editors," *ACM Transactions on Information Systems*, Vol.8, No.3, pp.237-268, Jul, 1990.
- [75] L. Verlet, "Computer Experiments on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules," *Physical Review*, Vol.159, No.1, pp.98-103, 1967.
- [76] G. Vriend, "WHAT IF: A Molecular Modeling and Drug Design Program," *Journal of Molecular Graphics*, Vol.8, pp.52-56, Mar, 1990.
- [77] J. Walton, "Visualization of Sphere Packs Using a Dataflow Toolkit," *Journal of Molecular Graphics*, Vol.12, pp.275-281, Dec, 1994.
- [78] J. Wernecke, *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Addison-Wesley, 1994.
- [79] J. Wernecke, *The Inventor Toolmaker: Extending Open Inventor*, Addison-Wesley, 1994.
- [80] R. Wirfs-Brock and R. E. Johnson, "A Survey of Current Research in Object-Oriented Design," *Communications of the ACM*, 33(9), pp.104-124, 1990.

BIOGRAPHICAL SKETCH

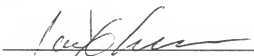
Jaeick Oh was born in Pusan, Korea, on August 10, 1961. He received a B.S. degree in control and instrumentation engineering from Seoul National University, Seoul, Korea in 1984. After graduation, he worked as a researcher at Gold Star Information and Systems R & D Laboratory in Seoul, Korea, from 1984 to 1990. His area of research included hardware and system software design for industrial control systems, communication software design for financial transaction systems.

He received an M.S. degree in computer and information science from the University of Florida in 1993. Currently, he is working toward a Ph.D degree in the Department of Electrical and Computer Engineering at the University of Florida. His current research interests are object-oriented software design, computer graphics, and scientific software system such as molecular modeling and visualization.

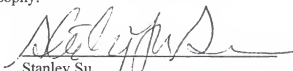
I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.


John Staudhammer, Chairman
Professor of Electrical and Computer
Engineering

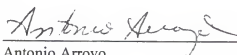
I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.


Paul W. Chun, Cochairman
Professor of Biochemistry and
Molecular Biology

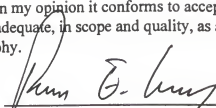
I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.


Stanley Su
Professor of Electrical and Computer
Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.


Antonio Arroyo
Associate Professor of Electrical and
Computer Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.



Panos E. Livadas

Assistant Professor of Computer and
Information Science and Engineering

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

December 1996



Winfred M. Phillips
Dean, College of Engineering

Karen A. Holbrook
Dean, Graduate School